BubbleStorm: Rendezvous Theory in Unstructured Peer-to-Peer Search

BubbleStorm: Rendezvous-Theorie in Unstrukturierten Peer-to-Peer Netzwerken
Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von B.Sc. Wesley W. Terpstra aus Victoria (Kanada)
2015 — Darmstadt — D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Databases and Distributed Systems

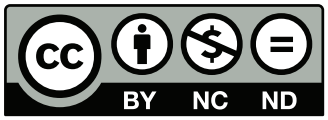
BubbleStorm: Rendezvous Theory in Unstructured Peer-to-Peer Search
BubbleStorm: Rendezvous-Theorie in Unstrukturierten Peer-to-Peer Netzwerken

Genehmigte Dissertation von B.Sc. Wesley W. Terpstra aus Victoria (Kanada)

1. Gutachten: Professor Alejandro Buchmann, Ph.D.
2. Gutachten: Professor Dr. Jussi Kangasharju
3. Gutachten: Dr. Ken Moody

Tag der Einreichung: 25. Nov 2014
Tag der Prüfung: 23. Jan 2015

Darmstadt — D 17



Contents

1 Introduction

The benefits of peer-to-peer are compelling, but so too are the problems. What developer wouldn't prefer a system where resources automatically increase with the user population? Community projects like Wikipedia or Debian with no clear revenue stream could benefit from the relief of infrastructure burden. Given these potential advantages, one must wonder why there are so few successful peer-to-peer applications compared to the Internet at large.

Still today, implementation of peer-to-peer applications remains the domain of network specialists and researchers. A very likely explanation for this is that existing peer-to-peer middleware requires immense expertise to use. Until peer-to-peer can abstract away the details of networking minutiae and provide a clean and reliable interface, it will remain inaccessible to the larger developer population.

In a way, the recent cloud computing trend targets the same niche: no private infrastructure and automatic load scaling. However, unlike existing peer-to-peer systems, the tech savvy companies backing these projects abstract away the network, aiming for an interface along the lines of distributed databases. Until peer-to-peer middleware makes this same leap, cloud computing will continue to dominate this niche and the losers will be non-profit community-driven projects and those whose content is unpopular among cloud administrators.

If we want peer-to-peer systems to be as successful as databases, we should look at what databases provide. First, they allow developers to define rich application-specific data models in the form of table schemas. Second, they provide a powerful declarative query language. Queries select elements from tables (or even cross-products of tables) using some predicate. To optimize queries, databases also track statistics used to plan their evaluation strategy. Finally, databases provide strong consistency guarantees. Today's mainstream peer-to-peer systems have none of these features, despite some ambitious attempts, like Willow [85], Astrolabe [84], and Minerva [58].

Established peer-to-peer search techniques can be roughly categorized into two models. The black-box model, followed for example by Gnutella [42] and Gia [17], does not process queries at the network level; it ignores their contents. Instead it ships the query to the data in the system for local evaluation. The key-value model, as exemplified by Chord [75], Pastry [70], and Kademlia [56], attaches to every query a key range which is used to route the query to only the data which has a matching key.

The problem with the key-value model is that it supposes that all queries are key lookups. This is obviously not the case and substantial work layers other query types atop the key-value infrastructure [15, 52, 69, 87]. Unfortunately, this layering trades away the performance gains from using keys and must be custom built for each new query type. Furthermore, it is not clear that the performance gained from using keys is always relevant. While the number of messages in key-value routing schemes is sometimes reduced, the routing depth remains $O(\log n)$, like most black-box schemes. As long

as the required bandwidth does not over-utilize the network, there is no latency penalty to the black-box scheme. Indeed, all the above cited approaches, which layer atop the key-value interface, pay not only a bandwidth penalty, but also a latency penalty for their added complexity. This trade-off is inappropriate for interactive searches where response times are critical.

As our goal is to make peer-to-peer applications easier to program, we will need great flexibility in the queries we support. In databases, a query scans the entire table looking for matches. While the system might employ optimizations behind the scenes, the application developer is in principle isolated from these concerns. To achieve comparable flexibility, we need to support the scenario where no search optimization is possible. The black-box model neatly captures this; it requires every query reach every data element. This has immediate advantages in simplicity and flexibility:

First, the black-box model is a very easy to understand abstraction. The network promises to ship your query to some peer which stores a copy of the data you seek. You can now solve your search problem locally, without needing to worry about the network. Developers don't need to be network experts; they just need to implement the semantics of their specific search problem, classically, on a single machine.

The simple black-box promise also allows us great freedom. A developer can re-use existing libraries to implement his query locally. For example, in some of our lab courses, the applications students built made heavy use of SQL. However, implementing our own SQL database engine would be a lot of work. Instead of tackling this task ourselves, we were able to just leverage the open-source SQLite [64] project. It implements SQL locally to each peer, but the black-box abstraction easily turns this into a distributed database[1]. If we had not been able transform a local solution into a distributed one, we would have had to re-implement a custom SQL engine.

While the black-box model is very flexible, this comes at a cost. Shipping a query to all data in the system is more bandwidth intensive than key-based routing schemes. The simplest approach is to flood the query to all participants, as discussed in any introductory text [74]. Obviously, this is also the least scalable, a serious problem for Internet-scale systems. Some approaches, like percolation search [72], restrict themselves to certain classes of graphs and then exploit properties of those graphs to expedite search. Unfortunately, these assumptions are very unrealistic and will be rejected in Chapter 3. Another approach, taken by Gia [17] and Cohen [24], relaxes the requirement of finding all hits to finding only the most popular hits. While cheaper than flooding, neither of these restricted approaches retain the full flexibility of the black-box model. This thesis will precisely quantify the bandwidth needed to retain the full flexibility of the black-box model and find that it is much cheaper than one might expect.

Peer-to-peer systems must not only manage search, but also storage. In the peer-to-peer model, nodes may crash at any time. Therefore, replication is typically used to ensure availability of the data. Of course, this raises a host of consistency issues, many of which remain unresolved. Many existing practical systems, for example Kademlia [56], republish information periodically to try to restrict the inconsistency window. Other ap-

[1] Of course, this trick only supports SELECT statements run over a single table. To support joins, more advanced techniques are needed; see our plans for full SQL support [50].

proaches, like Knežević's [46], make assumptions which preclude their use in real peer-to-peer systems. However, as we shall see, in every peer-to-peer replication strategy strong consistency is impossible, and these approaches fail to provide any guarantees in real systems.

It has been shown by Gilbert and Lynch [33] that a distributed system cannot simultaneously provide consistency and availability while tolerating network partition. In this context, availability is the ability to respond to queries and updates without blocking (perhaps indefinitely). Due to their scale, peer-to-peer systems are in a state of continuous failure and must therefore tolerate connectivity disruptions. These disruptions can partition a peer from the internet at large, violate graph connection transitivity, or even isolate an entire country. To remain useful, peer-to-peer systems must continue to function while these partition events (continuously) occur. Therefore, they must trade away consistency for availability. Nevertheless, this concession does not excuse peer-to-peer from failing to leverage the lessons learned from database success.

This thesis presents BubbleStorm [79], which attempts to bridge the gap between peer-to-peer and databases. BubbleStorm is a peer-to-peer search system, which solves large-scale rendezvous problems over the unreliable global internet. It provides a concept of user-defined bubble types, loosely corresponding to table schemas. Queries follow the fully general black-box model, allowing powerful queries to be evaluated exhaustively. The system tracks usage statistics with a system-wide measurement service (Chapter 8), both analogous to and useful in implementing a database's catalogue [50]. These statistics are used to automatically tune search performance. As strong consistency guarantees are impossible, BubbleStorm instead aims for user-controlled probabilistic guarantees.

The key contribution of this thesis is to develop rendezvous theory and reformulate the black-box query model within this framework. As we shall see, this formulation allows us to interpret any black-box system as solving a rendezvous problem, first recognized in my earlier BitZipper work [78]. This realization allows an elegant and tight lower-bound on any such system [77]. Independent groups [17,28,72] working on rendezvous systems in parallel to the development of BubbleStorm also recognized the need to relax BitZipper's strong consistency, but seem unaware of the theoretical underpinnings constraining their work. BubbleStorm is the only system leveraging rendezvous theory to substantially reduce bandwidth consumption (both practically and asymptotically) while simultaneously improving query latency. The resulting system, which has a full fledged implementation (Chapter 4), sports a simple to understand interface, which abstracts away the underlying details, much like the database systems before it.

2 Rendezvous Theory

Consider queries of the form "who sells X for under Y dollars?". If data elements look like "Z sells X for Y dollars", then peers can easily match a query against a data element, given they receive both. This is the heart of rendezvous systems.

We get much of the black box model's flexibility because the network is blind to the contents of queries, while the application is not. Queries could be XPath expressions, SQL select statements, or Java bytecode, but whatever they are, the application knows how to execute them. When the network is blind to the contents of queries, it cannot decide which data elements match. Only the local application can decide. As a network blind to query contents is already unable to match queries to data elements, we lose nothing by extending the black-box model to also blind the network to data. That data could be XML documents, SQL tuples, or binary strings, but again the application must know how to run queries on it.

When both queries and data are black-boxes, the only difference at the network layer is that the data elements are stored, while queries are processed and discarded. More generally, however, the query might be persistent as well. In publish-subscribe systems [27], the query is a subscription which watches for event notifications. Here the notifications are transient while the query is persistent. As another example, consider a query that first returns its result and then watches for changes. In this case, both the query and the data are persistent. Since both query and data are optionally persistent black-boxes, the network sees them as interchangeable. To reflect this symmetry, we refer to peers who process a query as having received a replica of the query.

Replication of data elements is already necessary in peer-to-peer systems to provide availability despite peer crashes. Once data is replicated, however, a natural benefit is that the load to serve that data is shared. A slightly more subtle point is that this replication pushes data closer to the peers who initiate queries. We will see much later (Section 9.4) that this improves query latency, while this chapter will show how this reduces bottleneck utilization.

Recall that in the black-box model, only the local application can decide if a query matches a data element. To ensure that a query q finds matching data element d, the network must guarantee that some peer receives both a replica of q and a replica of d. Then the application can locally open the boxes to evaluate if there is a match. For a given query-data pair, peers who can locally test for a match are the *rendezvous peers* (Figure 2.1). As the network cannot know which queries will match which data elements, it must ensure rendezvous for every query-data pair. Let the function $R(q) \subseteq U$ denote the subset of peers (out of all peers U) who receive a replica of query q and $R(d)$ the set of peers storing a replica of data element d. Then correctness in rendezvous systems is,

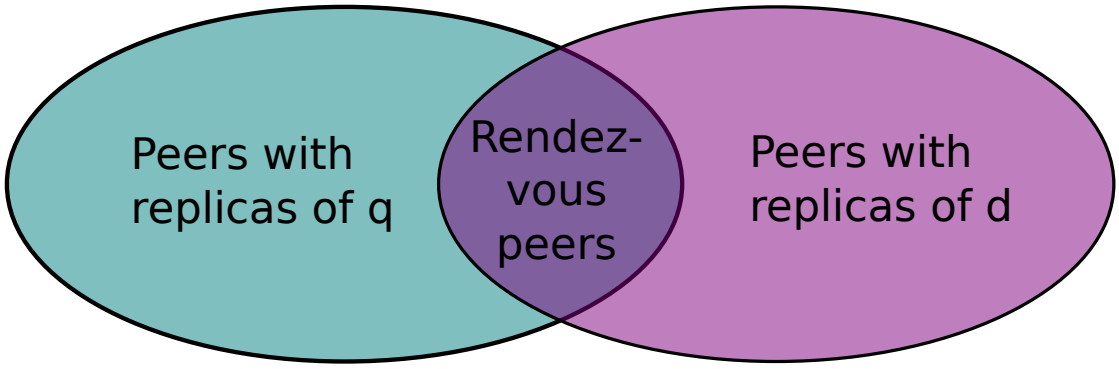


Figure 2.1.: Rendezvous of a query and data element

Definition 1 (The Rendezvous Problem)**.** *For a given set of documents D and queries Q, R is a valid replication function only if it guarantees the existence of a rendezvous peer for each query-data pair.* $\forall(q,d) \in Q \times D : R(q) \cap R(d) \neq \emptyset$

2.1 Bandwidth Metrics

Given a solution to the rendezvous problem, we need to be able to evaluate its bandwidth cost. There are a few metrics one might use, and these will be explored in this section. A perhaps surprising fact is that the bandwidth cost has little to do with the underlying graph connecting the peers. The underlying graph simply restricts which replication function R may be chosen.

In a complete rendezvous system, a search goes through three phases. First, the query is replicated out to peers in the network. Second, the peers locally evaluate the query for potential results. Finally, those results are returned to the peer initiating the query. The first and third phases incur a bandwidth cost.

This chapter will focus on the first cost, how many query replicas are required. The third cost, while important, has nothing to do with the rendezvous system, but everything to do with the specific query. The response traffic required is the query's result set, whose size remains the same in every correct algorithm. While reducing this bandwidth by dynamically selecting only some of the search results is an interesting problem, which the real BubbleStorm system addresses using Top-K and incremental search techniques, this thesis leaves these orthogonal issues to others.

When evaluating query replication cost, we need to know the size of both query and data replicas. Let S_q and S_d denote the size in bytes of a replica of $q \in Q$ and $d \in D$ respectively. We must also account for all required intermediate traffic. If a peer u sends d to v via the peer w, then $R(d)$ must include u, v, and w regardless of whether or not peer w actually bothers to store d. For convenience, we will define $S_D := \sum_{d \in D} S_d$ and $S_Q := \sum_{q \in Q} S_q$ as the total workload injected into the system (counted before replication).

Probably the most obvious metric is aggregate bandwidth,

Definition 2 (Aggregate Bandwidth Metric)**.** *Total traffic seen by the system.*

$$M_{aggregate}(R) = \sum_{d \in D} S_d |R(d)| + \sum_{q \in Q} S_q |R(q)|$$

Using this metric we can already see that flooding queries is often, but not always, a poor choice. The cost of flooding an n-peer system is,

$$M_{aggregate}(R) = S_D + S_Q n$$

If S_D is much larger than S_Q, flooding may perform quite well. For example, if the data elements were large multimedia files, then flooding a small search would be reasonable. Of course, it would probably be wiser to match the queries against the multimedia file's meta-data, rather than the entire file.

The problem with flooding is that as n grows, so do both S_D and S_Q. Usually each new peer adds his own traffic, so this growth is linear. This means the flooding term $S_Q n$ increases aggregate bandwidth quadratically, a clear scalability concern.

Aggregate bandwidth is an inappropriate metric for our work because the optimal algorithm under this metric is rendezvous at a central server. Set $R(q) = R(d) = \{u\}$ for all q and d. Certainly the rendezvous function is correct in the sense of Definition 1, but this result seems rather underwhelming. On the other hand, if the central server can handle the entire network's load, then this is unquestionably the most bandwidth efficient algorithm.

Given that the problem with the aggregate bandwidth metric is that it fails to capture the benefit of spreading the load between peers, let's try again. First, we will need a concept of peer capacity, C_u. If peer u can serve twice as many requests as v per unit time, then $C_u = 2C_v$. For this definition we will need the indicator function I_x which is 1 when x is true and 0 otherwise.

Definition 3 (Bottleneck Metric)**.** *Utilization of the most loaded peer.*

$$M_{bottleneck}(R) = \max_u \frac{1}{C_u} \left(\sum_{d \in D} S_d I_{u \in R(d)} + \sum_{q \in Q} S_q I_{u \in R(q)} \right)$$

Intuitively, in a perfectly load balanced system, the bottleneck metric measures the network-wide load. When it is 0.5, then the network is half utilized. Once the bottleneck metric exceeds 1, the network can no longer process the workload. Decreasing an implementation's bottleneck metric thus serves to increase the manageable workload. An implementation which is optimal with respect to the bottleneck metric can thus process the largest possible workload. In comparison, a system optimized for the aggregate bandwidth metric will probably cap out at a much lower potential workload (because it placed the entire workload on a few overloaded systems).

Under this new metric the central server scores,

$$M_{bottleneck}(\text{Central-Server}) = \frac{1}{C_u}(S_D + S_Q)$$

which can even be worse than flooding if S_D is large,

$$M_{bottleneck}(\text{Query-Flooding}) = \max_u \frac{1}{C_u}(S_D/n + S_Q)$$

The bottleneck metric is quite nice and we will use it when proving lower-bounds (Section 2.4) and analyzing static systems (Section 2.2). However, this thesis is about BubbleStorm, a probabilistic system. To measure its cost we need a slight variation on Definition 3 to account for its randomized nature.

The problematic terms in the bottleneck metric are the indicator functions. We cannot say for certain whether or not a peer receives a replica, so we cannot assign a simple zero or a one. In our final metric, we substitute the indicator with the probability that the event is true. This still ranges between zero and one, but captures the probabilistic nature of the system. In fact, the probability $\mathbf{P}(X)$ of an event X, is equal to the expected value of the indicator $\mathbf{E}(I_X)$. Therefore, this metric expresses the maximum expected utilization in the network. For a deterministic system, this is exactly the same as the simple bottleneck metric.

Definition 4 (Bottleneck Expectation Metric)**.** *The highest expected utilization amongst all peers.*

$$M_{expected}(R) = \max_u \frac{1}{C_u}\left(\sum_{d\in D} S_d \boldsymbol{P}(u \in R(d)) + \sum_{q\in Q} S_q \boldsymbol{P}(u \in R(q))\right)$$

As a final note for those with a more mathematical background, the bottleneck expectation metric is not the expected maximum utilization. When load is injected uniformly at random into a system, there is an unlucky peer somewhere which receives a traffic burst, and thus the expected maximum utilization is usually near capacity (1). Fortunately, due to the law of large numbers, these local traffic bursts are always transient. The metric as formulated avoids these tricky (and uninteresting) details by evaluating the expectation inside the maximum function. This allows us to meaningfully measure the overall scalability of a randomized system.

2.2 Grid Formulation

This section presents a grid topology to demonstrate that the black-box rendezvous problem can be solved much more efficiently than by flooding or utilizing a central server. Consider the $r \times c$ graph in Figure 2.2 containing $n = rc$ peers.

To solve the rendezvous problem on this graph, we must define a replication function. Assign every data element an identifier in the range $[0, r)$, so that the load is more-or-less balanced. Similarly, assign every query an identifier from $[0, c)$. We now define R so that a data element d with identifier i is replicated to all the peers on row i. Similarly, queries with identifier j are replicated to column j. As every row intersects every column exactly once, we have $|R(q) \cap R(d)| = 1$ for all q, d. Therefore, R is a solution to the rendezvous problem.

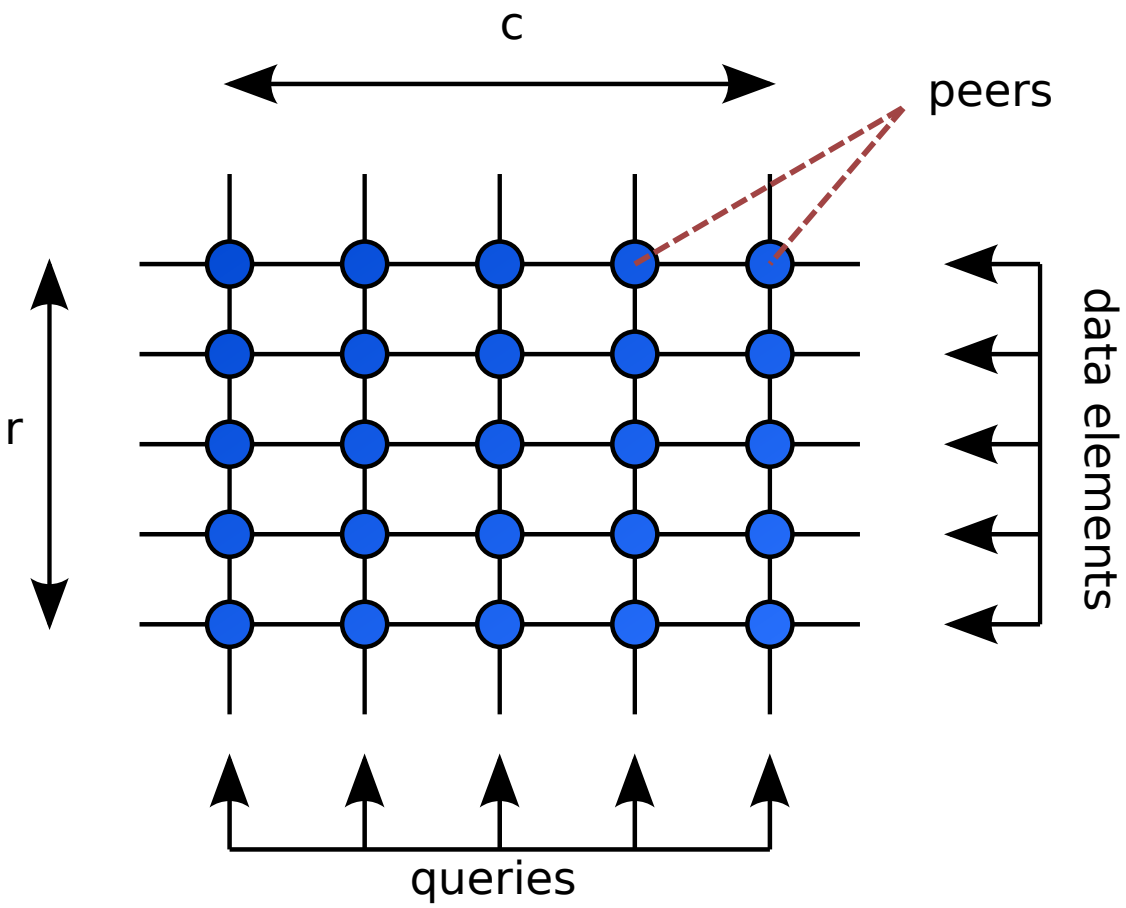


Figure 2.2.: A grid network topology for the rendezvous problem

Given that the data elements are load balanced over the grid rows, each row sees $\frac{1}{r}$ of the total traffic. The bottleneck cost in a homogeneous capacity grid is,

$$M_{bottleneck}(\text{Grid}) = \frac{1}{C_u}\left(\frac{S_Q}{c} + \frac{S_D}{r}\right)$$

One immediate benefit from using a grid is that we can control its shape. Choose $r = \sqrt{n\frac{S_D}{S_Q}}$ and $c = \sqrt{n\frac{S_Q}{S_D}}$. The grid still contains $rc = n$ peers (up to rounding). However, the cost then becomes,

$$M_{bottleneck}(\text{Grid}) = \frac{2}{C_u}\sqrt{\frac{S_Q S_D}{n}}$$

As the size of the query and data traffic grow linearly with n, this means that the cost grows as $\Theta(\sqrt{n})$. In other words, it fares one thousand times better than flooding or a central server on a one million peer network. In fact, we will see in Section 2.4 that with this choice of r and c, the grid approach uses exactly the minimum possible bandwidth (under the bottleneck metric).

Unfortunately, the grid approach is not practical. The core problem is that it cannot handle failure or adapt to change. In a peer-to-peer system, any number of peers can fail at any time; returning a match from exactly one peer is extremely fragile. Furthermore, as the system grows or shrinks, it would be very difficult to restructure the grid. Finally, the optimal choice of r and c depend on the ever changing traffic ratio of S_D to S_Q.

2.3 Poisson Formulation

A perhaps surprising result from probability theory is the Birthday Paradox. This result states it only takes 23 people to have a 50% chance that two of them share a birthday. More generally it takes $\Theta(\sqrt{n})$ people for n days. Building from this intuition, it hardly seems surprising that a grid solves the rendezvous problem. Indeed, it seems almost any scheme creating $\Theta(\sqrt{n})$ replicas should work. It is this insight which motivated the design of BubbleStorm.

The Poisson solution to the Rendezvous Problem simply places replicas uniformly at random onto peers in the network. Obviously, this is not guaranteed to always succeed. However, as discussed in the Introduction, it is impossible to build a peer-to-peer system which can answer queries both immediately and consistently. The chance that the Poisson solution fails is essentially the breach in consistency that we traded for queries which terminate.

Fortunately, as we shall shortly prove, the chance that the Poisson solution fails can be both calculated and controlled. There is a sharp boundary in the required number of replicas which takes us from almost certain failure to almost certain success. Unsurprisingly, this boundary is at $\Theta(\sqrt{n})$, just like the Grid formulation and the Lower Bound (Section 2.4). By increasing or decreasing the hidden constant, the success probability can be controlled according to the application's needs.

The Poisson formulation is both practical and elegant. It is hard to imagine a simpler procedure for placing replicas than random selection. The approach neatly avoids sticky issues like graph connectivity; compare to the Grid approach where each grid line must stay intact for the replication of query/data to succeed. As Theorem 1 will show, the Poisson approach still provides the best guarantee we could realistically hope for: a concrete and tunable success probability. There is no difficulty in "reshaping" the graph as the ratio of S_D to S_Q changes; we can just add or remove some replicas. For all of these reasons, the Poisson formulation is a very attractive solution in a peer-to-peer setting. BubbleStorm was designed to follow the heterogeneous version of this formulation (Section 2.5) very closely.

To analyze the failure probability of the Poisson approach, we use a simple physical analogy. Let the peers in the network each be a bin. Replicas are balls which are placed into these bins. A query replica is a blue coloured ball and a data replica is a red coloured ball. It can happen that one bin contains two red balls if the data was replicated to that peer twice. With Figure 2.3 in mind, it is easy to see that a search failure occurs when there is no bin containing both a red ball and a blue ball.

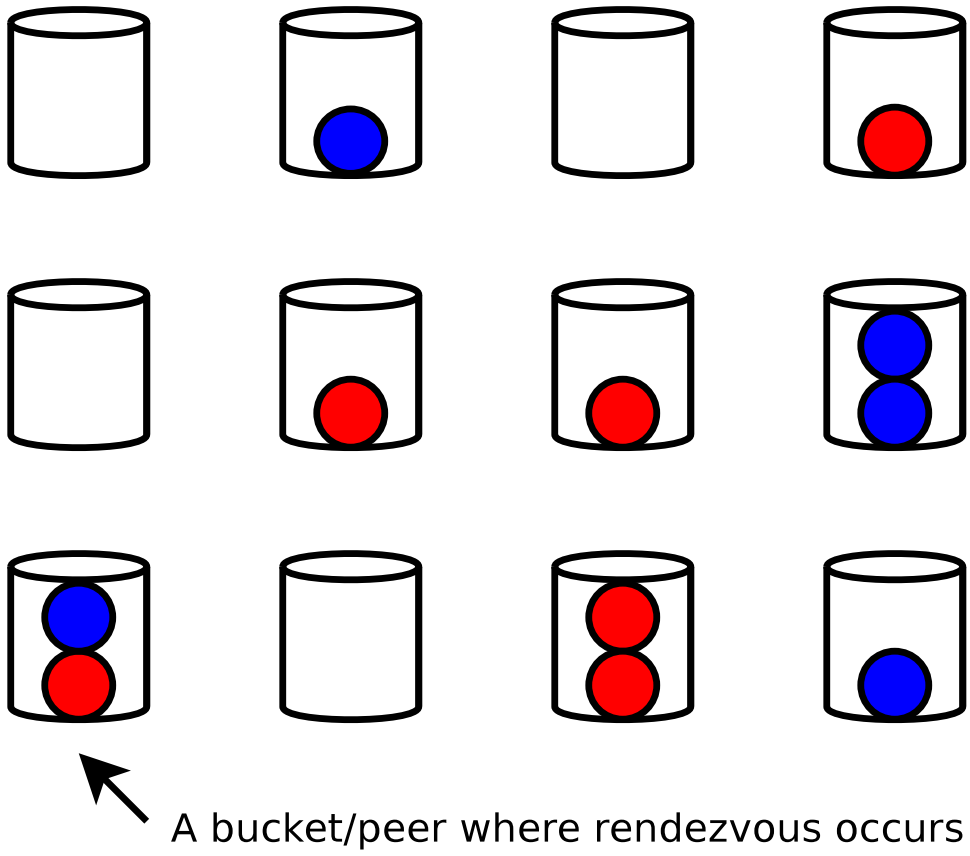


Figure 2.3.: Example of filling replicas (balls) into peers (bins)

2.3.1 Failure Probability

We will shortly prove the first main theorem. It is formulated using the definition $g(z) := 1 - e^{-z}$. This function safely approximates the concept of combining unlikely events. Given two independent 1% probability events, the chance of at least one event happening is almost (but not quite) 2%. For two 50% events, the chance is 75%, quite a bit below $50\% + 50\%$. $g(z)$ converts a straight-forward sum of event probabilities ($z = 0.5 + 0.5$) to a value, $g(z)$, guaranteed to be below the true result.

Theorem 1. *Place x blue balls and y red balls uniformly at random into n bins. Let M be the number of bins containing at least one blue ball and at least one red ball. Then, $\boldsymbol{P}(M = 0) \leq e^{-\lambda}$ whenever*

$$g(\lambda/n) \leq g(x/n)g(y/n)$$

To understand this relationship, realize that $g(z) \approx z$ for small z. Thus, this restriction is roughly $\lambda n \leq xy$, like in the Grid Formulation.

To prove this result we will first need two lemmas.

Lemma 2. *Let X be a discrete random variable, f a convex function, Y and Z indicator functions. If,*

- *X and Z are independent*
- *$\boldsymbol{P}(Y=1)=\boldsymbol{P}(Z=1)$*
- *$\boldsymbol{P}(Y=1|X=i)$ is decreasing in i*

Then,

$$\boldsymbol{E}(f(X+Y))\leq\boldsymbol{E}(f(X+Z))$$

Proof. Let $q_i:=\mathbf{P}(Y=1|X=i)$, and $p_i:=\mathbf{P}(X=i)$. Then $q:=P(Z=1)=\sum_i p_i q_i$. Since q_i is decreasing, find k such that $q_k\geq q\geq q_{k+1}$. Thus,

$$\begin{aligned}(q-q_i)&\leq 0 \quad \text{for } i\leq k\\ (q-q_i)&\geq 0 \quad \text{for } i>k\end{aligned}$$

f is convex; $f(i)-f(i+1)\geq f(i+1)-f(i+2)$. Induction yields,

$$\begin{aligned}f(i)-f(i+1)&\geq f(k)-f(k+1) \quad \text{for } i\leq k\\ f(i)-f(i+1)&\leq f(k)-f(k+1) \quad \text{for } i>k\end{aligned}$$

Subtracting the lemma's right-hand side and conditioning on X,

$$\begin{aligned}&\mathbf{E}(\mathbf{E}(f(i+Y)|X=i)-\mathbf{E}(f(i+Z)|X=i))\\ =\;&\sum_i p_i[f(i)(1-q_i)+f(i+1)q_i-f(i)(1-q)-f(i+1)q]\\ =\;&\sum_i p_i[f(i)(q-q_i)+f(i+1)(q_i-q)]\\ =\;&\sum_i p_i(q-q_i)[f(i)-f(i+1)]\\ =\;&\sum_{i\leq k} p_i(q-q_i)[f(i)-f(i+1)]+\sum_{i>k} p_i(q-q_i)[f(i)-f(i+1)]\\ \leq\;&\sum_{i\leq k} p_i(q-q_i)[f(k)-f(k+1)]+\sum_{i>k} p_i(q-q_i)[f(k)-f(k+1)]\\ =\;&(f(k)-f(k+1))\sum_i p_i(q-q_i)\\ =\;&(f(k)-f(k+1))\times 0\\ =\;&0\end{aligned}$$

□

Lemma 3. *Let X be the numbers of bins filled by placing x balls uniformly at random into n bins. Then for any convex function f, $\boldsymbol{E}(f(X)) \leq \boldsymbol{E}(f(\bar{X}))$, where $\bar{X}$ has a binomial distribution $B(n, h(x))$ and*

$$h(x) := 1 - \left(1 - \frac{1}{n}\right)^x$$

Proof. For every bucket $i \in [0, n)$, let B_i indicate whether or not that bucket has been filled with a ball. Define the partial sum,

$$S_j := \sum_{i=0}^{j-1} B_i$$

as the total number of filled buckets out of the first j buckets. Clearly, $X = S_n$. Set $\bar{S}_j$ to have a binomial distribution $B(j, h(x))$.

We will show by induction that $\mathbf{E}(f(S_j)) \leq \mathbf{E}(f(\bar{S}_j))$ for all $j \in [0, n]$ and convex f, thereby showing that $\mathbf{E}(f(X)) \leq \mathbf{E}(f(\bar{X}))$. The claim for S_0 is vacuously true. Assume the claim for S_j.

Create an indicator C independent of all other variables with $\mathbf{P}(C = 1) = \mathbf{P}(B_{j+1} = 1) = h(x)$. A bucket is not filled when all x balls failed to land in it. Thus,

$$\mathbf{P}(C = 1) = 1 - \left(1 - \frac{1}{n}\right)^x$$

We need to show that $\mathbf{P}(B_{j+1} = 1 | S_j = s)$ is decreasing in s. s balls are already used to fill the first j buckets. That leaves $x - s$ balls to place into $n - j$ potential buckets. $B_{j+1} = 0$ only if all balls placed fail to land in it. Thus,

$$\mathbf{P}(B_{j+1} = 1 | S_j = s) = 1 - \left(1 - \frac{1}{n-j}\right)^{x-s}$$

Define $f_c(x) = f(x + c)$ and notice that for all c, f_c remains convex. Now apply Lemma 2 followed by the induction hypothesis,

$$\begin{aligned} \mathbf{E}(f(S_{j+1})) &= \mathbf{E}(f(S_j + B_{j+1})) \leq \mathbf{E}(f(S_j + C)) = \mathbf{E}(\mathbf{E}(f_C(S_j)|C)) \\ &\leq \mathbf{E}(\mathbf{E}(f_C(\bar{S}_j)|C)) = \mathbf{E}(f(\bar{S}_j + C)) = \mathbf{E}(f(\bar{S}_{j+1})) \end{aligned}$$

□

We are now ready to prove the main Theorem.

Proof of Theorem 1. Let X denote how many bins received one (or more) of the x red balls, respectively Y for the y blue balls. Counting possibilities,

$$f(X, Y) := \mathbf{P}(M = 0 | X, Y) = \binom{n-Y}{X} / \binom{n}{X} = \prod_{i=0}^{Y-1} \left(1 - \frac{X}{n-i}\right)$$

Take the second derivative of $f(X,Y)$ with respect to X,

$$\frac{\partial}{\partial X}\frac{\partial}{\partial X}f(X,Y)=\sum_{i=0}^{Y-1}\frac{1}{n-i}\sum_{j=0,j\neq i}^{Y-1}\frac{1}{n-j}\prod_{k=0,k\neq i,k\neq j}^{Y-1}\left(1-\frac{X}{n-k}\right)\geq 0$$

Therefore f is convex in X when Y is held fixed.

Apply Lemma 3 to find,

$$\begin{aligned}
\mathbf{P}(M=0|Y) &= \mathbf{E}(f(X,Y)|Y)\\
&\leq \mathbf{E}(f(\bar{X},Y)|Y)\\
&= \sum_{i=0}^{n}\binom{n}{i}h(x)^i[1-h(x)]^{n-i}f(i,Y)\\
&= \sum_{i=0}^{n}\binom{n-Y}{i}h(x)^i[1-h(x)]^{n-i}\\
&= [1-h(x)]^Y\sum_{i=0}^{n-Y}\binom{n-Y}{i}h(x)^i[1-h(x)]^{(n-Y)-i}\\
&= [1-h(x)]^Y
\end{aligned}$$

Take the second derivative of $[1-h(x)]^Y$ with respect to Y,

$$\frac{\partial}{\partial Y}\frac{\partial}{\partial Y}[1-h(x)]^Y=(\ln[1-h(x)])^2[1-h(x)]^Y\geq 0$$

Now complete the bound by applying Lemma 3 again,

$$\begin{aligned}
\mathbf{P}(M=0) &= \mathbf{E}(\mathbf{E}(f(X,Y)|Y))\\
&\leq \mathbf{E}([1-h(x)]^Y)\\
&\leq \mathbf{E}([1-h(x)]^{\bar{Y}})\\
&= \sum_{j=0}^{n}\binom{n}{j}h(y)^j[1-h(y)]^{n-j}[1-h(x)]^j\\
&= \sum_{j=0}^{n}\binom{n}{j}\big(h(y)[1-h(x)]\big)^j[1-h(y)]^{n-j}\\
&= \big(h(y)[1-h(x)]+[1-h(y)]\big)^n\\
&= [1-h(x)h(y)]^n
\end{aligned}$$

To finish the proof, notice that $1-\frac{1}{n}\leq e^{-1/n}$ implies $g(z/n)\leq h(z)$. Now apply the Theorem's assumption that $g(\lambda/n)\leq g(x/n)g(y/n)$,

$$1-e^{-\lambda/n}=g(\lambda/n)\leq g(x/n)g(y/n)\leq h(x)h(y)$$

And thus,

$$\mathbf{P}(M=0)\leq[1-h(x)h(y)]^n\leq e^{-\lambda}$$

□

2.3.2 Limit Results

We are interested in how the Poisson approach to the rendezvous problem scales. Here we let $n \to \infty$ and ask interesting questions like:

- How many replicas are needed for a target failure rate $e^{-\lambda}$?
- What is the optimal trade-off between data and query replication?
- What is the distribution of the number of matching responses?

Theorem 1 goes a long way towards answering the first two of these questions. Take the Taylor series approximation to the constraint,

$$\begin{aligned}
\frac{\lambda}{n} - \frac{\lambda^2}{2n^2} + O\left(\frac{1}{n^3}\right) &= 1 - e^{-\lambda/n} \\
&\leq \left(1 - e^{-x/n}\right)\left(1 - e^{-y/n}\right) \\
&= \left(\frac{x}{n} + O\left(\frac{1}{n^2}\right)\right)\left(\frac{y}{n} + O\left(\frac{1}{n^2}\right)\right) \\
&= \frac{xy}{n^2} + O\left(\frac{1}{n^3}\right)
\end{aligned}$$

We can thus conclude that as $n \to \infty$, the constraint becomes $\lambda n \leq xy$. Therefore, the number of data and query replicas must grow as $O(\sqrt{n})$, as we had already anticipated.

Just like the grid formulation, the product xy allows us to trade query replication against data replication. Recalling the terms S_D and S_Q, the total data and query workload in bytes, we can find the optimal trade-off. Set $x = \sqrt{\lambda n \frac{S_D}{S_Q}}$ and $y = \sqrt{\lambda n \frac{S_Q}{S_D}}$. Now the total traffic is $S_Q x + S_D y = 2\sqrt{\lambda n S_D S_Q}$, and the cost in a homogeneous capacity network becomes,

$$M_{expected}(\text{Poisson}) = \frac{2}{C_u}\sqrt{\lambda \frac{S_Q S_D}{n}}$$

Compared to the Grid formulation, we have introduced a factor $\sqrt{\lambda}$. This factor scales the cost up while simultaneously scaling the failure probability down. When $\lambda = 1$ for parity with the grid formulation, the failure probability is 36.8%. If we double the cost as compared to the grid, we achieve $\lambda = 4$ and 1.8%. Triple the cost for 0.01% or 99.99% success rate. As should be clear, a relatively small increase in λ rapidly reduces the failure chance.

To answer the last question, how many times query and data meet, we analyze $\mathbf{P}(M = k)$. As we shall shortly prove, this distribution is Poisson in the limit with rate λ, justifying my choice to call this approach to the rendezvous problem the Poisson formulation. What this means in practice is that for $\lambda = 4$, one expects 4 peers to receive both the query and data. For some examples of the Poisson distribution, see Figure 2.4.

We now prove the Poisson nature of the Poisson formulation,

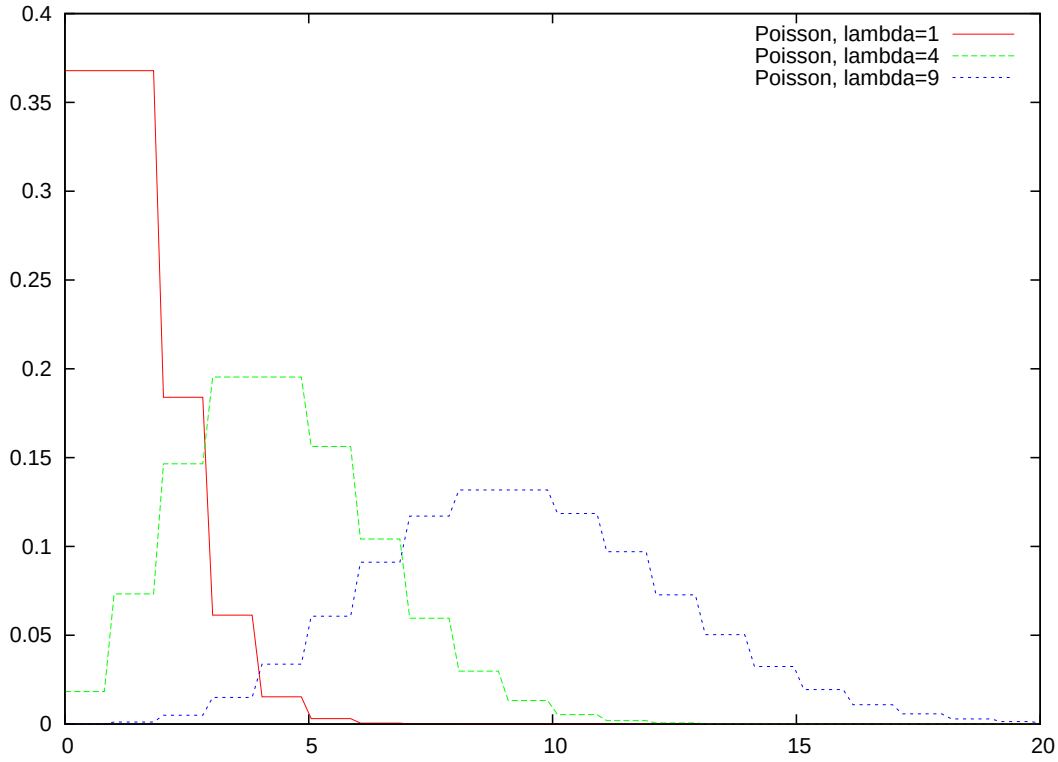


Figure 2.4.: Examples of the Poisson distribution; the x-axis is the number of peers where data and query meet and the y-axis shows the probability.

Theorem 4. *Place x blue balls and y red balls uniformly at random into n bins. Let M be the number of bins containing at least one blue ball and at least one red ball. When $n \to \infty$, $\frac{x}{n} \to 0$, and $\frac{y}{n} \to 0$, then $\mathbf{P}(M = k) \to \frac{\lambda^k}{k!} e^{-\lambda}$ whenever*

$$g(\lambda/n) = g(x/n)g(y/n)$$

Proof. Since $\frac{x}{n} \to 0$ and $\frac{y}{n} \to 0$, we will operate under the assumption that $n-x-y > 0$. Observe that $X \leq x$ and $Y \leq y$ by definition to find,

$$\begin{aligned}
\mathbf{P}(M = k|X,Y) &= \binom{Y}{k}\binom{n-Y}{X-k}/\binom{n}{X} \\
&= \frac{1}{k!}\frac{X!}{(X-k)!}\frac{Y!}{(Y-k)!}\frac{(n-X-Y)!}{(n-X-Y+k)!}\frac{(n-X)!(n-Y)!}{n!(n-X-Y)!} \\
&= \frac{1}{k!}\frac{X!}{(X-k)!}\frac{Y!}{(Y-k)!}\frac{(n-X-Y)!}{(n-X-Y+k)!}\mathbf{P}(M = 0|X,Y) \\
&\leq \frac{1}{k!}\left(\frac{XY}{n-X-Y}\right)^k \mathbf{P}(M = 0|X,Y) \\
&\leq \frac{1}{k!}\left(\frac{xy}{n-x-y}\right)^k \mathbf{P}(M = 0|X,Y)
\end{aligned}$$

We will need a new helper function h, with $h(z) \to 1$ as $z \to 0$. By l'Hôpital's rule,

$$h(z) := \frac{z}{1-e^{-z}} \to \frac{1}{e^{-z}} \to 1$$

Finally, take the expectation and apply Theorem 1,

$$\begin{aligned}
\mathbf{P}(M=k) &\leq \frac{1}{k!}\left(\frac{xy}{n-x-y}\right)^k \mathbf{P}(M=0) \\
&\leq \frac{1}{k!}e^{-\lambda}\left(\frac{xy}{n-x-y}\right)^k \\
&= \frac{1}{k!}e^{-\lambda}\left(\frac{xy}{n-x-y}\right)^k \frac{\lambda^k}{\lambda^k}\frac{\left(1-e^{-\lambda/n}\right)^k}{\left(1-e^{-x/n}\right)^k\left(1-e^{-y/n}\right)^k}\frac{n^k n^k}{n^k n^k} \\
&= \frac{\lambda^k}{k!}e^{-\lambda}\left(\frac{1}{1-x/n-y/n}\right)^k \left[h(x/n)\right]^k\left[h(y/n)\right]^k \frac{1}{\left[h(\lambda/n)\right]^k} \\
&\to \frac{\lambda^k}{k!}e^{-\lambda}
\end{aligned}$$

As we have now bounded $\mathbf{P}(M=k)$ in the limit by the Poisson distribution, it follows that $\mathbf{P}(M=k) \to \frac{\lambda^k}{k!}e^{-\lambda}$. Otherwise, $\mathbf{P}(M=x) \to \frac{\lambda^x}{x!}e^{-\lambda} - \epsilon$ for some x and $\epsilon > 0$ would contradict that $\sum_k \mathbf{P}(M=k) = 1$. □

2.4 Heterogeneous Lower Bound

As promised, we now turn our attention to proving a lower-bound on the complexity of the rendezvous problem. We call a solution to the rendezvous problem R correct whenever it satisfies Definition 1, that is $R(q) \cap R(d) \neq \emptyset$ for all q, d. The bound this section proves applies only to correct solutions.

As discussed in Section 2.1, the lower-bound also depends on the metric. When we use the simple aggregate traffic metric, $M_{aggregate}$ from Definition 2, the best we can show is that $M_{aggregate}(R) \geq S_D + S_Q$. The proof below will work exclusively with the bottleneck metric, $M_{bottleneck}$ from Definition 3.

Theorem 5. *Any correct solution to the rendezvous problem R must obey,*

$$M_{bottleneck}(R) \geq 2\sqrt{\frac{S_D S_Q}{\sum_{u \in U} C_u^2}}$$

Before we dive into the proof, a high-level intuitive explanation may help. In Figure 2.5 the grey area represents the pairs of queries and data; like in the Grid formulation the queries are denoted by the x-axis and the data by the y-axis. A given peer can only match queries against data it has received. In the example figure, peer u downloads all the queries in its image on the x-axis and all the data in its image on the y-axis.

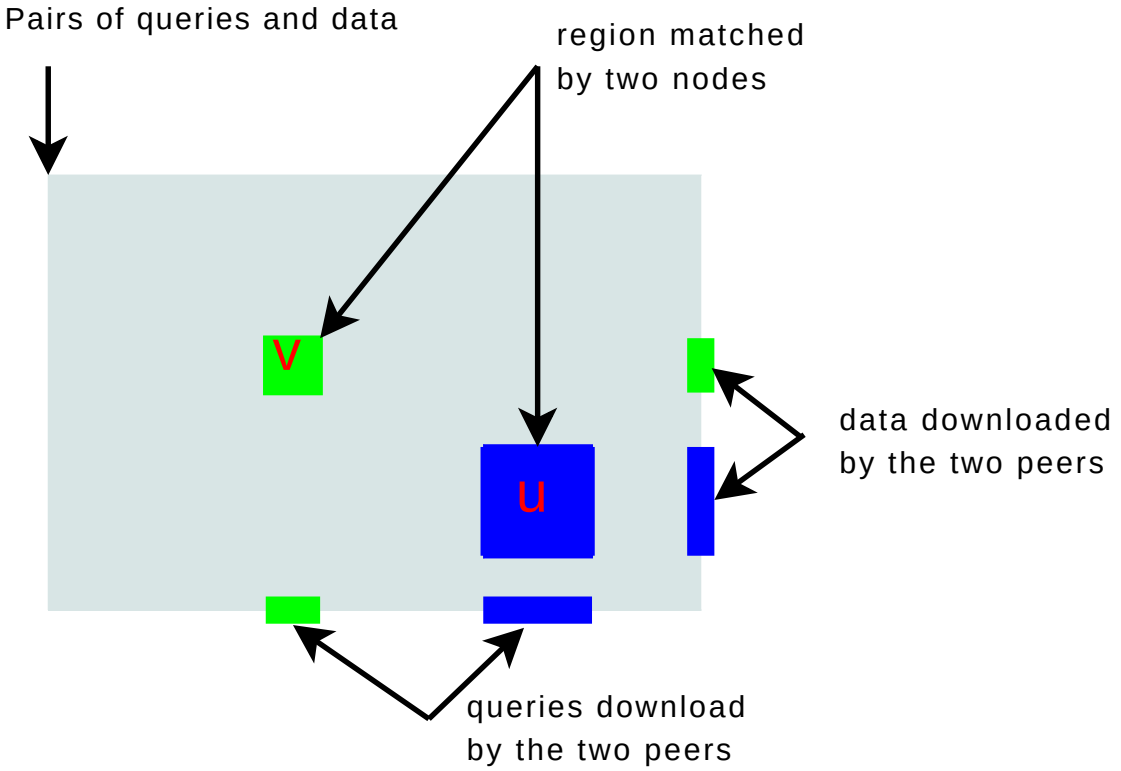


Figure 2.5.: A visual representation of the lower-bound

For a solution to the rendezvous problem to be correct, all of the grey area (pairs) in the figure must be covered by some peer.

The amount a peer matches is obviously related to the area it covers. A peer who downloads twice as many queries and twice as much data can match four times as many pairs. In this way, the rendezvous processing power of a peer depends on the square of its capacity. In the figure, this is why peer u covers four times the area of peer v, despite the fact that it only downloads twice as much. The processing power of the entire network is thus $\sum_{u \in U} C_u^2$ and it serves to process the workload area of size $S_Q S_D$, intuitively justifying the ratio in Theorem 5.

Generally speaking, given a fixed circumference the rectangle which maximizes its area is a square. The sum of the height and width of a square is twice the square-root of the square's area, explaining the outer part of Theorem 5. The proof below shows this more formally.

Proof of Theorem 5. The correctness constraint assures us that for every pair $(q, d) \in Q \times D$ the intersection $R(q) \cap R(d)$ is non-empty. Therefore,

$$\begin{aligned} \exists u \in U: \quad & u \in R(q) \cap R(d) \\ \exists u \in U: \quad & I_{u \in R(d)} I_{u \in R(q)} && = 1 \\ \exists u \in U: \quad & S_d I_{u \in R(d)} S_q I_{u \in R(q)} && = S_d S_q \end{aligned}$$

Summing over all peers in the network,

$$\sum_{u \in U} S_d I_{u \in R(d)} S_q I_{u \in R(q)} \geq S_d S_q$$

As this relationship holds for all pairs $(q, d) \in Q \times D$,

$$\begin{aligned} \sum_{u \in U} \left(\sum_{d \in D} S_d I_{u \in R(d)} \right) \left(\sum_{q \in Q} S_q I_{u \in R(q)} \right) &= \sum_{(q,d) \in Q \times D} \sum_{u \in U} S_d I_{u \in R(d)} S_q I_{u \in R(q)} \\ &\geq \sum_{(q,d) \in Q \times D} S_q S_d = S_Q S_D \end{aligned} \tag{2.1}$$

To complete the proof, suppose for contradiction that,

$$\begin{aligned} 2\sqrt{\frac{S_D S_Q}{\sum_{u \in U} C_u^2}} &> M_{bottleneck}(R) \\ &= \max_{u \in U} \frac{1}{C_u} \left(\sum_{d \in D} S_d I_{u \in R(d)} + \sum_{q \in Q} S_q I_{u \in R(q)} \right) \end{aligned}$$

It would follow that for every $u \in U$,

$$2C_u \sqrt{\frac{S_D S_Q}{\sum_{u \in U} C_u^2}} > \sum_{d \in D} S_d I_{u \in R(d)} + \sum_{q \in Q} S_q I_{u \in R(q)}$$

When $2x > a + b$ with $a > 0$ and $b > 0$, the maximum value for ab is x^2. Set $a = \sum_{d \in D} S_d I_{u \in R(d)}$ and $b = \sum_{q \in Q} S_q I_{u \in R(q)}$. Then,

$$\left(\sum_{d \in D} S_d I_{u \in R(d)} \right) \left(\sum_{q \in Q} S_q I_{u \in R(q)} \right) < C_u^2 \frac{S_D S_Q}{\sum_{u \in U} C_u^2}$$

Sum over all peers,

$$\sum_{u \in U} \left(\sum_{d \in D} S_d I_{u \in R(d)} \right) \left(\sum_{q \in Q} S_q I_{u \in R(q)} \right) < \sum_{u \in U} C_u^2 \frac{S_D S_Q}{\sum_{u \in U} C_u^2} = S_D S_Q$$

And now we have our contradiction when compared to line 2.1. □

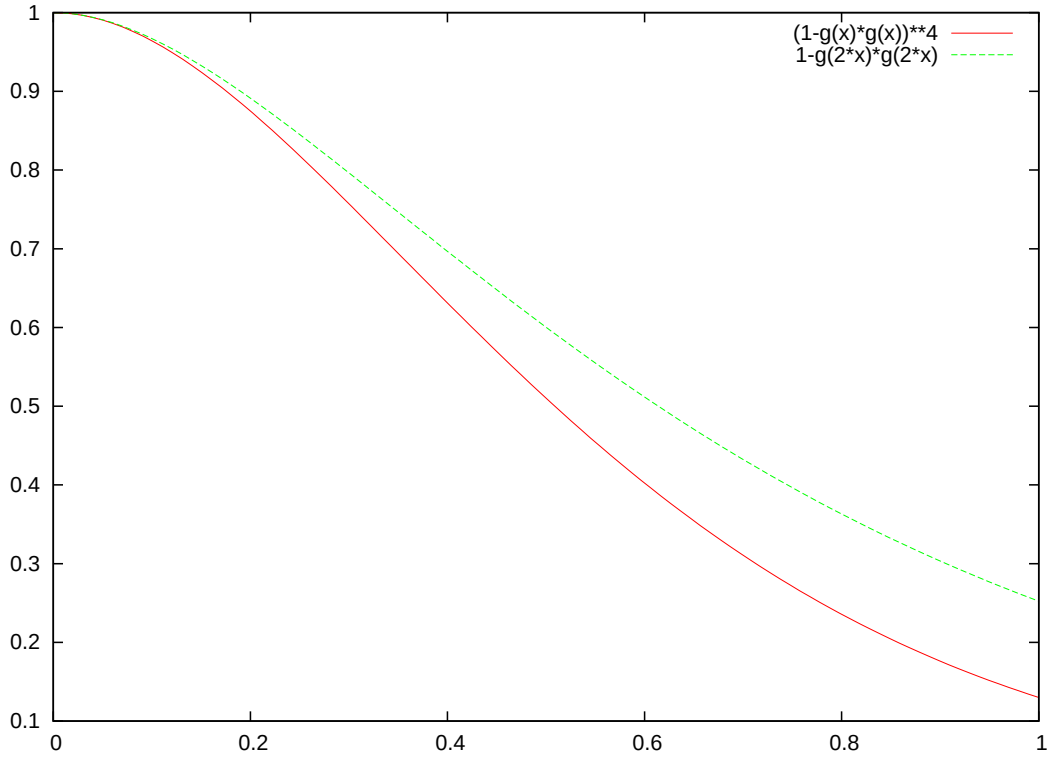


Figure 2.6.: Failure probability of a single double capacity peer as compared to four normal capacity peers as a function of the expected number of balls received by the normal capacity peer.

2.5 Heterogeneous Formulation

We have seen in the lower-bound proof that higher capacity peers can be leveraged to decrease the overall utilization of the network. This section describes a generalization of the Poisson formulation which can exploit this potential.

Assign every peer u a weight $w_u = C_u / \sum_v C_v$, so that $\sum_u w_u = 1$. Whenever we place a ball, peer u receives that ball with probability w_u. This model can exploit heterogeneity by making double capacity peers twice as likely to receive work.

Unfortunately, there is a slight wrinkle to this plan, illustrated in Figure 2.6. While doubling the probability to receive a ball does double the expected number of balls a peer receives, it does not quite double the probability of non-zero balls. If w_u is twice w_v, it is true that u is twice as likely as v to receive a particular ball. However, it does not follow that when more than one ball is placed, u is twice as likely as v to receive *some* ball. This is the same phenomenon, combining unlikely events, that motivated the definition of g in Theorem 1. As there, we are again on the wrong side of the inequality.

Fortunately, as long as z is small, $g(z)$ is close to z. Therefore, this phenomenon is only a problem when z is large. This corresponds to the situation where we are placing so many balls of one particular colour that the chance that a particular bucket receives

that ball is high. In other words, it becomes a problem when either (or especially both) of these conditions are true:

- one peer/bucket is significantly larger than all others
- one colour ball saturates the network, presumably because the other replica type is rare; presumably because it is significantly larger, skewing S_D/S_Q

BubbleStorm must still work correctly (i.e., meet $e^{-\lambda}$) in these situations, though perhaps with slightly degraded bandwidth performance. Theorem 1 already handled the case of skewed S_D/S_Q by using $g(x/n)$ instead of x/n. Consequently, we must simply remain vigilant in our handling of heterogeneity. We must ensure that if a single peer dominates the network, we keep its influence on the correct side of the $g(z)$ approximation function.

That said, we can now state the main theorem which we prove later,

Theorem 6. *Place x blue balls and y red balls into bins with probability w_u for bin u. Let M be the number of bins containing at least one blue ball and at least one red ball. Then,*

$$\boldsymbol{P}(M=0) \leq \prod_{u \in U} \big(1 - g(w_u x) g(w_u y)\big)$$

This result is the heterogeneous analogue of Theorem 1. Set $w_u = 1/n$ and $|U| = n$ to derive Theorem 1 from Theorem 6.

The problem with Theorem 6 is that it is not very useful in practice. In a peer-to-peer setting, it would be very expensive to obtain all values of w_u, which requires knowing the capacity of every peer in the network. This makes solving for x and y in Theorem 6 problematic. Fortunately, there is an easy to compute upper-bound which is also a good approximation.

We have seen that bigger peers behave slightly worse than a corresponding number of smaller peers, due to the combination of unlikely events. Thus, we can't easily pull the w_u out of the protective $g(w_u x)$. However, the inequality only prevents us from extracting large coefficients out of g; we *can* push a larger value inside. This motivates us to approximate all terms in Theorem 6 by the equivalent expression for the largest peer in the network. Since $g(z) \approx z$ for small z, the common case where no peer receives nearly all the traffic, we expect this approximation will be asymptotically good.

Theorem 7 (The BubbleStorm Balance Equation)**.** *Place x blue balls and y red balls into bins with probability w_u for bin u. Let w_m be the maximum value of w_u for all $u \in U$. Let M be the number of bins containing at least one blue ball and at least one red ball. Then $\boldsymbol{P}(M=0) \leq e^{-\lambda}$ whenever,*

$$g\left(\lambda \frac{w_m^2}{\sum_{u \in U} w_u^2}\right) \leq g(w_m x) g(w_m y)$$

This formula is much easier to work with. It only requires knowing the maximum capacity peer in the network and the sum of the capacities squared. In Chapter 8 we will see that both of these quantities are easy to obtain using a peer-to-peer measurement protocol. The approximation in Theorem 7 will be used by BubbleStorm's convex optimizer in Chapter 5. The derivation from Theorem 6 is straight-forward,

Proof. We first apply Lemma 9 to Theorem 6 with $k = w_m/w_u$,

$$\begin{aligned}
\mathbf{P}(M=0) &\leq \prod_{u\in U}\left(1-g(w_u x)g(w_u y)\right) \\
&= \prod_{u\in U}\left(1-g(w_u x)g(w_u y)\right)^{k^2\frac{w_u^2}{w_m^2}} \\
&\leq \prod_{u\in U}\left(1-g(w_m x)g(w_m y)\right)^{\frac{w_u^2}{w_m^2}} \\
&= \left(1-g(w_m x)g(w_m y)\right)^{\sum_{u\in U}\frac{w_u^2}{w_m^2}}
\end{aligned}$$

Now we apply the Theorem's assumption,

$$\begin{aligned}
\mathbf{P}(M=0) &\leq \left(1-g(w_m x)g(w_m y)\right)^{\sum_{u\in U}\frac{w_u^2}{w_m^2}} \\
&\leq \left(1-g\left(\lambda\frac{w_m^2}{\sum_{u\in U}w_u^2}\right)\right)^{\frac{\sum_{u\in U}w_u^2}{w_m^2}} \\
&= \left(e^{-\lambda\frac{w_m^2}{\sum_{u\in U}w_u^2}}\right)^{\frac{\sum_{u\in U}w_u^2}{w_m^2}} \\
&= e^{-\lambda}
\end{aligned}$$

□

Since BubbleStorm uses Theorem 7, we will prove that this approximation remains optimal, rather than proving optimality for Theorem 6. Since Theorem 7 is just an upper-bound on Theorem 6, we will also have shown optimality for Theorem 6. It might be surprising that after making so many approximations the utilization remains optimal. However, in each step, we have only made approximations which are sharp in large networks.

Theorem 8 (Optimality of BubbleStorm)**.** *Whenever* $w_m \in o(\sqrt{\sum_{u\in U} w_u^2})$,

$$M_{expected}(BubbleStorm) \to 2\sqrt{\lambda\frac{S_Q S_D}{\sum_u C_u^2}}$$

Theorem 8 assumes that the contribution of the largest capacity peer vanishes. This means we assume that as peers are added to the network, the size of the largest peer grows slower than the combined capacity of the network. When this assumption is not met, BubbleStorm remains correct (as it satisfies Theorem 7), but fails to meet the lower-bound for optimal/minimal traffic.

This restriction makes a lot of sense. If a single peer is so large that it performs most of the matching, it is unable to download equal parts query and data. It behaves instead like a server, downloading everything, and thus the ratio of downloaded query replicas to data is dictated by the workload. We have seen in the lower-bound that optimality requires peers to download equal parts data and query replicas. The restriction ensures that this is possible.

Proof. We have already seen in the proof of Theorem 4 that l'Hôpital's rule shows $g(z) \approx z$ for small z. Under our assumption that $w_m \in o(\sqrt{\sum_{u \in U} w_u^2})$,

$$g\left(\lambda \frac{w_m^2}{\sum_{u \in U} w_u^2}\right) \to \lambda \frac{w_m^2}{\sum_{u \in U} w_u^2}$$

Furthermore, we know that x and y grow as $O(1/\sqrt{\sum_{u \in U} w_u^2})$. Thus,

$$\begin{aligned} g(w_m x) &\to w_m x \\ g(w_m y) &\to w_m y \end{aligned}$$

In the limit, the BubbleStorm correctness constraint (Theorem 7) becomes,

$$\lambda \frac{w_m^2}{\sum_{u \in U} w_u^2} \leq w_m x w_m y$$

The optimizer (Chapter 5) can do no worse than picking,

$$\begin{aligned} x &= \sqrt{\lambda \frac{S_D}{S_Q} \frac{1}{\sum_{u \in U} w_u^2}} \\ y &= \sqrt{\lambda \frac{S_Q}{S_D} \frac{1}{\sum_{u \in U} w_u^2}} \end{aligned}$$

Now find the aggregate traffic,

$$M_{aggregate}(\text{BubbleStorm}) = S_Q x + S_D y = 2\sqrt{\lambda \frac{S_Q S_D}{\sum_{u \in U} w_u^2}}$$

Each peer v receives traffic with probability w_v. Therefore, its expected utilization is,

$$M_{expected}(\text{BubbleStorm}) = \frac{1}{C_v} w_v M_{aggregate}(\text{BubbleStorm}) = 2\sqrt{\lambda \frac{S_Q S_D}{\sum_{u \in U} C_u^2}}$$

Since this is the same for all peers, the proof is complete. □

To finish this section, we now prove the two results we needed earlier.

Lemma 9. *For $k \geq 1, x \geq 0, y \geq 0$,*

$$[1-g(x)g(y)]^{k^2} \leq 1-g(kx)g(ky)$$

Proof. The Hölder generalized mean inequality states,

$$\sqrt[1]{(1-z)p^1+zq^1} \leq \sqrt[k]{(1-z)p^k+zq^k}$$

for $k \geq 1$ and $0 \leq z \leq 1$. Set $p = 1$ and $q = e^{-x}$. Then,

$$\begin{aligned}\left(1-zg(x)\right)^k &= \left((1-z)1+ze^{-x}\right)^k \\ &\leq (1-z)1^k+ze^{-kx} \\ &= 1-zg(kx)\end{aligned}$$

Apply this inequality twice with $z = g(x)$ and $z = g(ky)$,

$$\begin{aligned}\left(1-g(x)g(y)\right)^{k^2} &\leq \left(1-g(x)g(ky)\right)^k \\ &\leq 1-g(kx)g(ky)\end{aligned}$$

□

Proof Sketch for Theorem 6. To show

$$\mathbf{P}(M=0) \leq \prod_{u\in U}\left(1-g(w_u x)g(w_u y)\right)$$

Notice that $M = 0$ only if no bucket received both a red and blue ball. Let I_u indicate that peer u did not receive both types of ball. Now, $I_{M=0} = \prod_u I_u$. If buckets $I_1, I_2, ..., I_k$ have not received both ball types, it implies there are more balls to place in the $(k+1)^{th}$ bucket; $\mathbf{P}(I_{k+1} = 1 | I_1 I_2 ... I_k = 1) < \mathbf{P}(I_{k+1} = 1)$. Thus,

$$\mathbf{P}(M=0) \leq \prod_{u\in U}\mathbf{P}(I_u = 1)$$

The chance that a bucket receives a red ball is,

$$\mathbf{P}(u \text{ receives red}) = 1-(1-w_u)^x \geq g(w_u x)$$

The chance it receives both ball types is,

$$\mathbf{P}(u \text{ receives both}) \geq g(w_u x)g(w_u y)$$

Therefore,

$$\begin{aligned}\mathbf{P}(M=0) &\leq \prod_{u\in U}\mathbf{P}(I_u = 1) \\ &= \prod_{u\in U}\mathbf{P}(u \text{ receives neither}) \\ &\leq \prod_{u\in U}(1-g(w_u x)g(w_u y))\end{aligned}$$

□

2.6 Summary

Now that all the important rendezvous theory results are proven, this section summarizes them for quick reference:

- Rendezvous systems should be designed to minimize bottleneck utilization.
  - This maximizes the workload the network can process.
  - This automatically balances the workload amongst peers.
- Peer bandwidth capacity pays off in the square.
  - It is critical to leverage high capacity peers.
  - If one peer dominates, the system degenerates to a centralized server.
- Rendezvous systems have a simple and elegant lower-bound.
  - One can meaningfully discuss an implementation's optimality.
- Where replicas are placed is relatively unimportant.
  - Careful placement can meet the lower-bound exactly, but ...
  - Random placement of those replicas already results in one match on average.
- Random placement is very practical for peer-to-peer systems.
  - Placing exactly x balls (allowing duplicates in buckets) works.
  - Each bucket could also flip a $g(x/n)$-weighted coin to obtain a ball.
  - The resulting number of rendezvous peers is Poisson distributed.
  - The failure probability is easily controlled (Theorem 7) and sharply approximates optimal utilization (Theorem 8).

3 Related Work

Some of the earliest work related to rendezvous systems was done on quorum systems. Quorum systems are used in classical databases to provide consistent replication [32, 40, 83], mutual exclusion/locking [3, 53], read/write registers [5, 54], and group communication [4, 13]. While these all appear to be very different from the rendezvous problem, the core mathematical principle is the same: find sets which contain a non-empty intersection.

Originally, quorum systems were studied to ensure that commits in a distributed system were well ordered. The first approaches used a majority vote [83] to decide that a change was committed. Thus, any future update which also achieved a majority would have to have encountered the previous commit. Of course, all that is required is existence of what we call a rendezvous peer, a result quickly formalized in [30]. From there it was an easy step to the grid formulation [19], although there were even earlier approaches [53] that achieved $O(\sqrt{n})$ complexity.

Just like rendezvous systems, quorum systems have a well defined lower-bound that applies to any correct quorum algorithm. In [63], the authors even use the same metric we choose for BubbleStorm, the maximum expected utilization. They conclude, as we do, that the lower-bound is a square-root.

The CAP principle [33] applies equally to quorum systems. Thus, a quorum system has to choose at most two of: consistency / correctness, availability / operations that terminate, or tolerance of network partition. The quorum systems above picked consistency and partition tolerance. However, one can also pick availability and partition tolerance. This approach is taken by [55].

In peer-to-peer, a randomized approach to quorum has its advantages. In [55], they follow an approach similar to that of BubbleStorm. Each quorum consists of $\sqrt{\lambda n}$ *distinct* replicas. They show that in such a scenario, the chance that two quorums fail to intersect is $e^{-\lambda}$. Furthermore, they show that the introduction of probabilistic guarantees allow them to achieve near optimal load and resilience simultaneously. Thus, they see a probabilistic correctness guarantee as a means to bypass the CAP principle, the same motivation that drove us to build BubbleStorm after BitZipper.

If one views BubbleStorm as a probabilistic quorum system, then we further the field in five areas. First, quorum systems do not account for heterogeneity. As we have already shown in Section 2.4, there are substantial performance improvements available when one leverages the asymmetric power of participating peers. Concretely, compare Figure 5.8 to 5.9. Second, we consider placing replicas with potential duplicates and/or coin-flip (binomial) distribution. Both of these are more realistic models than placing exactly $\lambda\sqrt{n}$ replicas. This practical concession does cost us the g-approximation. Third, we consider trading off the size of the intersecting bubbles; quorum systems have consistently concerned themselves with symmetric load. In a rendezvous system, it is quite certain that query and publish traffic loads will not match. As Section 5.4 and Figure 5.8

demonstrate, there are huge gains to be made in this area. Fourth, BubbleStorm considers more than simply read/write-quorums. Intersections between multiple bubble types are supported, a necessary feature for a rich data schema. Fifth, BubbleStorm is a complete system, not just a mathematical theory.

Returning our attention to the world of peer-to-peer, there are several other systems that attempt to solve the rendezvous problem. We will compare these systems with an eye towards their practicality. For an overview of rendezvous systems in other contexts, see Section 2.13 of my colleague's thesis [49].

Especially important for peer-to-peer systems are resilience and adaptability. In particular, systems must be able to handle crashing peers, packet losses, the inability of some pairs of peers to communicate, and large scale network outages. On the adaptability front, the system must adjust to changing network size, peer bandwidth distribution, and query/data traffic ratio. We consider systems in chronological order.

One of the first peer-to-peer rendezvous systems, Gnutella [42], actually scores very well on the requirements checklist. This very simple system floods searches to (nearly) all peers. Thus, it does not take advantage of the query/data traffic distribution. However, flooding is very resilient to losses. Gnutella continues to operate if a large fraction of the network is cut. Furthermore, as it is unstructured, it does not depend on complete connectivity. High capacity Gnutella peers simply establish more connections, so it adapts well to changing peer bandwidth distribution. In fact, Gnutella is pretty much a perfect peer-to-peer system. The only problem is that flooding queries does not scale.

On the complete opposite end of the spectrum, we have Google Grid [9]. While not a peer-to-peer system, we include it as it is the first published true rendezvous system. As it runs in a data center, it chooses availability and consistency, sacrificing the ability to operate with continuously failing peers. The google grid, as the name suggests, follows the grid formulation of the rendezvous problem. The entire database is "sharded" into pieces vertically. All the peers in a column contain between them the complete database. A single row in the grid duplicates a "shard" of data to all the peers in that row. Thus database updates are applied to rows and queries are applied to columns. Due to the static nature of the grid, it cannot easily be adapted to a changing number of member peers, heterogeneous peer capacities, or query/data traffic ratio. Nevertheless, it is highly efficient, representing an optimal solution to the rendezvous problem.

To the best of my knowledge, my BitZipper system [78] was the first to tackle the *balanced* rendezvous problem in a peer-to-peer setting. As a new researcher in the field of peer-to-peer, I was enamored with the idea of key-based routing overlays. Thus I made the mistake of proposing BitZipper on top of Chord [75], a structured peer-to-peer system. Consequently, BitZipper cannot tolerate incomplete connectivity between peers. All peers must be able to communicate with their ring neighbours, or routing breaks down. On the plus side, leveraging Chord did allow BitZipper to tolerate significant network outages and adapt to changing network size.

BitZipper operates on the principle of alternating bits in the keyspace. Every query and published data item are assigned a random identifier. In queries, every odd bit is turned into a wildcard. For published data, every even bit is wildcarded. Thus, for any given query+data pair, there is a unique key that matches both, corresponding to taking

bits from them alternatingly (zipping). Peers store a replica if they are responsible for a key which matches the wildcarded set. Thus, the node responsible for the zipped key receives a replica of both the query and data item. It is the rendezvous peer.

In the BitZipper system, due to the uneven responsibility ranges of peers in Chord, on average a query/publish is replicated to $2\sqrt{n}$ peers. Due to the need to route via intermediate peers, this grows to $6\sqrt{n}$. This overhead corresponds to $\lambda = 36$, enough to guarantee fifteen nines of reliability in BubbleStorm. Given that Chord fails with rate of 2% [34] even on a very reliable network with nearly no churn and no NATs, BitZipper performs far worse than BubbleStorm. BitZipper did offer the ability to statically trade-off query-data bandwidth via a fixed-length prefix. However, it could neither adapt to changing query/data ratios, nor leverage heterogeneous peers. All of these problems motivated us to start work on BubbleStorm.

Before we were finished, however, Ferreira et al. [28] published the first rendezvous system for unstructured peer-to-peer networks. Thus, they avoid the problems BitZipper has with incomplete connectivity. They use random walks of length $\sqrt{\lambda n}$ to replicate queries and publish data. Naturally, they concluded that this results in a failure rate of $e^{-\lambda}$. Unfortunately, *this is incorrect!* However, one cannot fault them too harshly, as we will see that all other related work repeatedly makes the same error.

The problem with their claim of $P(M = 0) \leq e^{-\lambda}$ is that they confuse placing replicas uniformly at random with distinct replicas. Their result would be true for distinct replicas, as [55] does for quorum systems. The result would even be true if they had all n peers flip a coin weighted by $\sqrt{\lambda n}/n$, as we do in BubbleStorm's maintained replication [51]. The proof techniques used in Theorem 1 demonstrate both these cases correct. However, placing replicas uniformly at random, as in the Ferreira system, does *not* work. An easy to check counter-example: $n = 2, \lambda = 8$. $\sqrt{n\lambda} = 4$. So, consider placing 4 query and 4 data replicas to avoid a rendezvous. All 4 queries must land on the same peer, 1/8 likely. All 4 data replicas must land on the other peer, 1/16 likely. $1/128 > e^{-8}$, contradicting the claim. Our use of g in the correctness constraint fixes this problem.

The Ferreira system did not include an actual implementation, nor propose an underlying network topology. Therefore, it is impossible to say how it would tolerate crashes, churn, outages, etc. It is clear from their simulation results that they simply ran a simulation according to their mathematical model. Unfortunately, in a real network, one must add a correction factor for topological dependency (Section 9.1), or there will be a systematic defect in the probability. As discussed, they also did not correctly factor in the effects of replica collisions, which necessitated the g-factor in Theorem 1. Thus, in a real system, they would not meet their target probability. Furthermore, random walks are terribly unreliable as compared to trees for replication (see [79] or Section 9.4). The system also completely ignores the issue of balancing query/data traffic. Finally, instead of leveraging heterogeneity to improve performance, they utilize a Metropolis-Hastings algorithm [6] to stamp it out.

Next up is our first BubbleStorm publication [79]. Designate this system as BubbleStorm0 to distinguish it from the correct BubbleStorm described in this thesis. BubbleStorm0 was the first peer-to-peer system which simultaneously tackled adaptability

to network size, bandwidth shape/composition, and query/data traffic balance. Furthermore, it is robust to network outages and incomplete network connectivity. The organization of the subsystems is quite similar to this thesis. However, we fell into the same trap as Ferreira et al., believing that random replica placement alone suffices for correctness. While we were aware of the error term in our correctness equation, we thought that it did not matter, despite that fact that it scaled with the cube; $e^{-\lambda+\lambda^{3/2}\epsilon}$. In our defense, the error term matters little when the query/data traffic are near parity and λ is small, so our first simulator could not measure the defect easily. BubbleStorm0 did, by virtue of the analysis done in [77], compensate for topological dependency, though not by name. What it got wrong was missing the g-approximation for collisions and incorrectly believing that heterogeneity pays off exactly as in the lower-bound. We explained in Section 2.5 why this fails to hold. Compared to the prototype BubbleStorm0, BubbleStorm is a complete re-implementation. The bubble balancer (Chapter 5) combined with Theorem 7 solve the correctness issues with BubbleStorm0, while the performance of both the topology and measurement protocol are superior.

Shortly after BubbleStorm0, ROAR was published [65–67]. Later versions of ROAR focus on an implementation for data centers, which we won't consider here; they are more related work for Google Grid than BubbleStorm. The version of ROAR proposed to run on key-based structured peer-to-peer is deterministic. It places data replicas along an arc of the ring, say r out of n peers. Queries then contact every r-th peer on the ring. Thus, they contact $c = n/r$ peers. Clearly, this is a slight variant of the grid formulation.

As a follow-up to my BitZipper work, ROAR improves the ability to balance query/data traffic. While BitZipper had a fractal replication scheme, the much simpler arc-oriented data replication scheme of ROAR allows it to increase/decrease r fairly easily. However, like BitZipper, ROAR builds on a structured system. This has the same immediate consequences in terms of robustness and waste.

Like BitZipper, ROAR has a unique rendezvous peer. Should the system fail to reach that peer or that peer fail, ROAR fails. Given that structured systems do not deliver reliably and fail to handle incomplete connectivity, these problems cannot be easily mitigated so long as a structured system underlies ROAR. Furthermore, this probability cannot be controlled like in BubbleStorm, and will likely be worse than $\lambda = 4$.

On the cost front, the key-based routing imposes overhead. While data placement only costs $\log n + r$, it involves an $r = \sqrt{n}$ random walk. Fortunately, this walk follows the ring, which typically has redundant edges to ensure integrity. It is unclear if ROAR leverages this redundancy to route around those edges which cannot be formed (due to incomplete connectivity) or are broken/crashing. However, ROAR *could* do something intelligent here and we give it the benefit of the doubt when compared to random walk schemes like Ferreira's. The queries, however, must pay a routing cost to reach the data replicas, to the tune of $c \log n$. While it might be possible to share the cost of messages routed over common edges, like BitZipper does, there is a limit as discussed in [78]. Regardless, this log factor will probably exceed BubbleStorm's homogeneous overhead at $\lambda = 4$. Coupled with BubbleStorm's use of heterogeneity, BubbleStorm will probably have orders of magnitude better reliability for the same traffic cost when compared to this "deterministic" solution.

My colleague, Christof Leng's thesis [49] also covers BubbleStorm. His thesis focuses on the replication algorithms we used to make data durable, a topic not covered here. With replication in hand, he is also able to directly compare BubbleStorm to Kademlia [57]. Despite being published first due to life and timing, his thesis builds on the work published here. In particular, he cites the correctness theorems and the bubble balancer from this thesis. Now that this thesis has been published, it closes these gaps in his work. Of necessity, he included a brief overview of the underlying layers described in detail here. However, the current topology and measurement protocols and their analysis only appear in this text. Whenever I refer to BubbleStorm in this thesis, I am referring to our combined work; it is not possible to compare/contrast these two versions of BubbleStorm as they are one and the same.

Finally, there have been a few rendezvous systems which snuck through the peer review process. For example, Hautakorpi [38] and Deetoo [20]. These won't be discussed here as they bring nothing new to the table. Indeed, both are strictly inferior to at least one of the prior art systems described here. Also, they both suffer from flawed analysis concluding that $\sqrt{\lambda n}$ replicas suffice to reach $e^{-\lambda}$. To find a more forgiving analysis, please refer to [49].

4 BubbleStorm Overview

BubbleStorm is a system which solves the black-box rendezvous problem. It's implementation both informed the development of rendezvous theory and followed the theoretical underpinnings discovered. The project has since evolved beyond this original black-box goal. However, my work has mostly focussed on the theory and implementation of rendezvous systems and this thesis chooses to focus on these aspects of BubbleStorm.

At its highest level of rendezvous abstraction, BubbleStorm provides the user with a concept of bubble types. A bubble type is category of information whose meaning is known to the user, but remains a black box to the system. As an example, a bubble type might be video meta-data, perhaps covering authorship, copyright, and length. Concrete bubbles of this type would correspond to particular videos. One bubble might describe someone's birthday video, including the requisite meta-data.

Bubbles take their name from the role they play in the system. A particular bubble (an instance of a bubble type) contains those peers who store its data. If that birthday video's meta-data is stored by five peers in the network, its bubble has size five. Bubble types are the means by which the user formulates his rendezvous problems and bubbles the vehicle which drives their operation.

Continuing our example, the home movie programmer might need to be able to locate videos by a particular author. The query template "find content by X" would comprise another bubble type, and a concrete search for Fred's movies a bubble. To ensure that the search succeeds in finding all of Fred's movies, we must guarantee rendezvous between author query and movie meta-data. The programmer thus tells the BubbleStorm system that meta-data type bubbles must intersect author search type bubbles. If the programmer has instructed BubbleStorm to intersect two bubble types A and B, then the system will guarantee rendezvous between all pairs $(a, b) \in A \times B$ with the programmer-specified rate λ. Using the notation from Section 2, $R(a) \cap R(b) \neq \emptyset$. This concept is so fundamental to BubbleStorm that both the book cover and first figure in this thesis (Figure 2.1) depict two bubbles intersecting.

To use BubbleStorm, the programmer must thus do five things. First, he defines his bubble types, in some sense specifying his application's data model. Then, he specifies which pairs of bubble types must intersect, enforcing correctness. He provides callback functions which are invoked to store bubbles in his application-specific database and process search requests. Next, he grows a network comprised of peers running his application. Finally, during the operation of his application, he blows bubbles with concrete data. These steps will now be covered in detail.

Continuing with our example, suppose that the application also stores blog articles. Then we have three types of bubble so far: video meta-data, blog articles, and authorship searches. Both the video meta-data and blog articles need to be stored persistently, while the search need only be processed to see if it matches the two persistent bubbles. When creating a bubble type, the programmer must select one of four bubble

INSTANT	Bubbles in this class are not stored persistently. This is generally only useful for queries.
FADING	These bubbles are stored persistently by peers, but no attempt is made to refresh the bubble as replicas are lost to churn. This suites data which has a natural expiration or is constantly changing. Examples might include the position of a space ship in a game.
MANAGED	These bubbles are stored persistently by peers, so long as their owner remains in the system. Once the owner leaves the system, the bubble evaporates. Until then, the bubble size ensures reliable rendezvous.
DURABLE	These bubbles are stored persistently by peers, indefinitely. They are managed by an epidemic protocol which ensures the stability of their size despite churn or changing balance.

Figure 4.1.: Bubble classes supported by BubbleStorm

classes from Figure 4.1. For every class except INSTANT, the programmer must specify a callback to store replicas of that type. For the MANAGED and DURABLE classes, he must also specify a few methods which can retrieve replicas, so they may be copied to other peers as the system evolves. For an SQL-backed bubble type, the storage callback probably just triggers an INSERT statement. In summary, then, the programmer creates each bubble type he will need, selecting a storage class and providing hooks for storing/retrieving replicas of this type.

Next, the programmer specifies which bubble types must intersect. In our example, the authorship search type must intersect both the video meta-data and blog article bubble types. When the programmer requests intersection, he specifies a desired rate λ and a processing callback. The system guarantees rendezvous between bubbles of the intersected type with a probability of at least $1-e^{-\lambda}$ and not much higher. In a perfectly Poisson world, the programmer would also expect λ rendezvous peers. Due to various complications, the real expectation may be slightly higher (due to a topology-correcting scale factor in Section 9.1) or lower (due to large single-response-only heterogeneous peers replacing multiple smaller peers in Section 2.5). As already discussed in the optimality of BubbleStorm (Theorem 8), $\lambda = 4$ costs twice the traffic of $\lambda = 1$.

The processing callback for an intersection is run to perform the rendezvous. When calling the API, `intersect(type1, type2, λ, callback)`, the types are not interchangeable. The callback is run when bubbles of type1 arrive, provided with their type1 content. If a callback should also be run upon the arrival of type2 bubbles (to build a persistent query, for example), then `intersect` should be called again with the types reversed. In our example, when the programmer specifies an intersection between authorship search and video meta-data, the callback probably runs an SQL SELECT statement over the video meta-data table using the author provided in the search bubble. When the programmer intersects authorship search with blog articles, he may

use a different λ and must provide another callback. This callback presumably runs a SELECT over the blog table.

There is a good reason intersection callbacks and storage callbacks are distinct. In our example, there will likely be other query types that should be run over the video meta-data. Furthermore, the blog articles and video metadata might be replicated to a different number of peers. Suppose the video meta-data bubbles are larger than blog article bubbles. Then, not every peer who receives an authorship query should run that query on its video metadata. Otherwise, the effective λ for the video metadata and author search would be too high. By specifying a separate match callback for each intersection, BubbleStorm has the freedom to invoke the blog matching callback and not the video matching callback when it receives a query replica. Thus BubbleStorm can freely balance bubble sizes how it likes and still meet the desired match probabilities as closely as possible.

4.1 Component Architecture

In Figure 4.2, the green areas are components which I worked on directly. The very green components will be covered in this thesis, while the reader is invited to read my publications for the light green components.

Here is a brief overview of the BubbleStorm stack,

- The Runtime component includes operating system specific hooks to perform task scheduling and UDP+ICMP messaging.
- The Simulator component simulates a virtual network with many peers running concurrently and communicating via virtual UDP+ICMP messages.
- The Channel-based Unidirectional Stream Protocol (CUSP) [82] implements a TCP work-alike on top of UDP+ICMP. It implements cryptography, flow control, congestion control, and reliable in-order delivery of multiplexed streams. We will depend on its ability to puncture firewalls in Section 7.2.1.
- The Network Topology layer manages the connections between peers in the network. It will be discussed in Chapter 7. A prototype was first described in [79].
- The Measurement Protocol is responsible for calculating global statistics used by all higher layers. It can compute sums and maxima and will be covered by Chapter 8. A poorer performing variation was briefly discussed in [79, 80].
- The Bubble Balancer is responsible for solving the Bubble Balance Equation (Theorem 7). It computes the number of replicas needed to ensure successful rendezvous in the network. It appears in Chapter 5.
- Bubblecast is the protocol we use to push replicas into the network. The replication algorithm is essentially unmodified from [79] and will be covered in Chapter 9, where the first analysis of its dependency correction will appear.
- Maintained Replicas ensure that replicas placed in the network last exactly as long as their owner, implementing the MANAGED bubble class. It was published in [51] and will not be covered here.
- Collective Replicas ensure that replicas not tied to a particular peer remain in the system, implementing the DURABLE bubble class. It also allows for $O(1)$ key-value lookups, eliminating the need for DHTs when using BubbleStorm. This work was published in my colleague's Doctoral thesis [49].
- Incremental and Top-K Search layer more sophisticated query techniques on top of the basic bubblecast primitive. This work is not yet published.
- The SQL component aims to implement an SQL database on top of BubbleStorm. It is published in [50], but not yet implemented.
- The full text search component is used for most test scenarios.

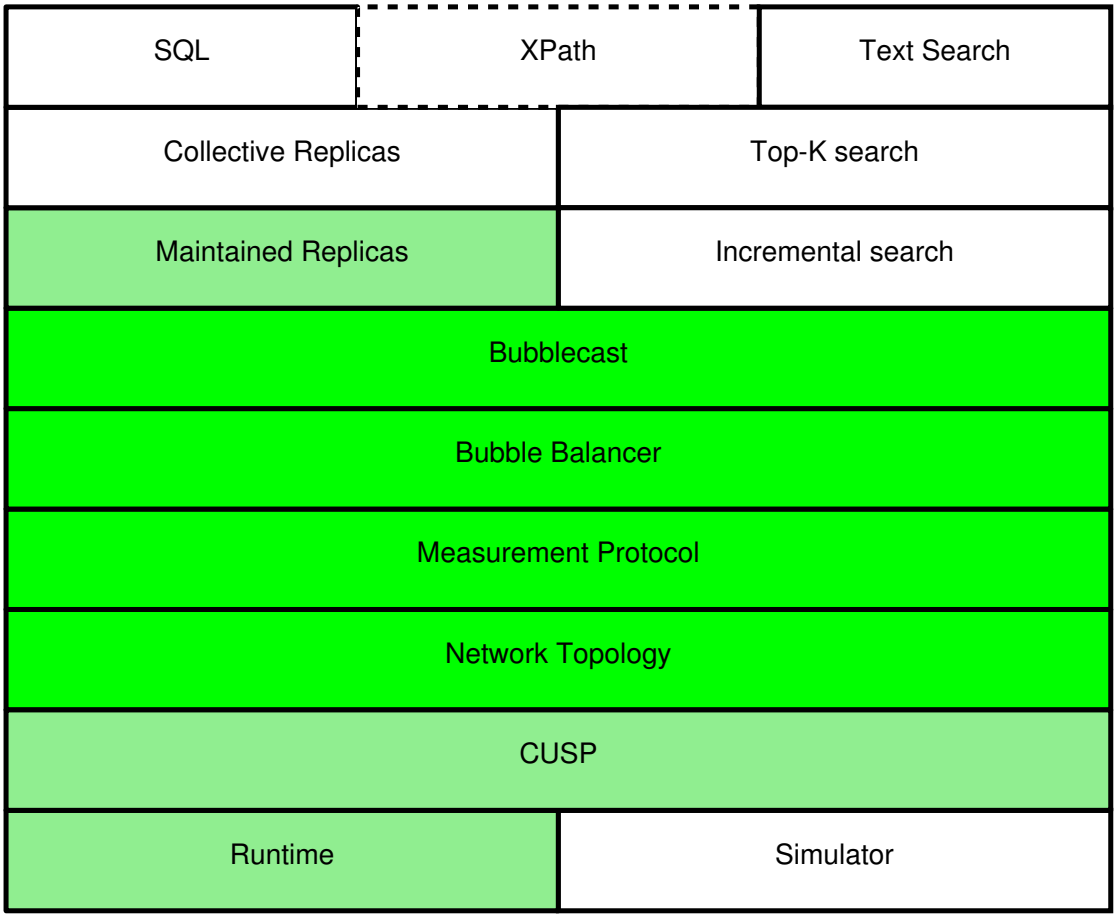


Figure 4.2.: Component diagram of the BubbleStorm system

While much of the focus of this thesis was covered in my earlier publications [77, 79–81], this thesis is the first publication to present a complete and correct unstructured rendezvous system. The development of BubbleStorm went through many false starts where our incomplete understanding of rendezvous theory led us to make poor design choices. Furthermore, our experience in developing robust peer-to-peer protocols has also evolved. While [79] presented a topology and measurement protocol quite similar to those presented here, the theory was missing and there were devils in the implementation details. The current BubbleStorm implementation works on real networks with protocols that provide much greater reliability and can be proven correct.

5 Bubble Balancer

The bubble balancer ensures that the rendezvous probability of all bubble-pairs is met, whilst minimizing the traffic needed. As a simple example, consider a query-data bubble-type pair. If there are many queries and the data changes rarely, it makes sense to store the data on many peers. Then the query need not be replicated to many peers. Thus, we trade a large data bubble for a small query bubble. This makes sense because the query bubble is used more heavily.

In the previous example, the traffic costs are just the S_D and S_Q from Section 2.1. However, in the actual BubbleStorm implementation, we allow many bubble types. Let T be the set of programmer-defined bubble types. We define S_t as the traffic (before replication) of bubble-type t.

As described in the system overview (Section 4), the programmer specifies λ, requiring BubbleStorm to guarantee a rendezvous with probability $\geq 1 - e^{-\lambda}$ between any chosen pair of bubble types $s, t \in T$. These bubble-type rendezvous requirements are, to borrow database terminology, part of the data model's schema for this application. Every peer in the network runs the same end-user application (or at least operates on the same schema) and was thus initialized with the same types T and the same intersection requirements $\{\lambda_{st}\}$.

The bubble balancer's job is thus to ensure that the bubble balance equation (Theorem 7) is met for every intersection requirement. Let x_t be the size (number of replicas for each instance) of bubble-type t. Then the correctness requirements can be stated as,

$$1 - e^{-\lambda_{st} \frac{w_m^2}{\sum_{u \in U} w_u^2}} \leq \left(1 - e^{-w_m x_t}\right)\left(1 - e^{-w_m x_s}\right) \text{ for all } s, t \in T \tag{5.1}$$

The total system traffic which the balancer seeks to minimize is,

$$\sum_{t \in T} S_t x_t \tag{5.2}$$

The goal of the algorithm is to find the x_t which minimize 5.2 subject to 5.1. The algorithm receives information about the bubble types, and their intersection requirements λ_{st}, from the programmer. The values w_u describing the shape of the network are obtained from the measurement protocol, which will be covered in Section 8. To find S_t, every peer u records the size (in bytes) of locally generated traffic for bubbles of type t. This is then summed over all peers $u \in U$ using the measurement protocol. The result is S_t, the traffic bubbles of type t would cost if only replicated once.

With all these inputs in-hand, the bubble balancer is left with an optimization problem. Luckily[1], the constraints in this optimization problem can be formulated convex. This allows us to leverage the field of convex optimization.

[1] By dint of carefully spent sweat-blood-and-approximation.

5.1 Convex Formulation

The BubbleStorm implementation includes a simple convex optimizer (Section 5.3). The optimizer accepts a linear goal function, of the same form as 5.2. The optimizer only supports constraints in terms of these convex functions,

$$ax + by + cz \geq 0 \tag{5.3}$$
$$x \geq e^y \qquad \text{for } |y| > 1 \tag{5.4}$$
$$x \geq e^y - 1 \qquad \text{for } |y| < 1 \tag{5.5}$$

I implemented two different exponential functions to handle the two different cases where the right-hand-side is very small. For $y \to -\infty$, $e^y \approx 0$ and for $y \approx 0$, $e^y - 1 \approx 0$. These two implementations of the exponential function are numerically stable and very accurate at these important limit points.

We now reformulate 5.1 in terms of the constraints 5.3 - 5.5. The first thing to notice is that the left-hand-side (LHS) of 5.1 is a constant. As ln is an increasing function, we can reformulate 5.1 as,

$$\ln LHS \leq \ln\left(1 - e^{-w_m x_t}\right) + \ln\left(1 - e^{-w_m x_s}\right) \tag{5.6}$$

The main trick to reformulating the bubble balance problem in terms our simple convex optimizer can handle is the introduction of carefully chosen intermediate variables. Here we substitute y_t for $\ln(1 - e^{-w_m x_t})$ in 5.6 and add a new requirement,

$$\ln LHS \leq y_t + y_s \tag{5.7}$$
$$y_t \leq \ln(1 - e^{w_m x_t}) \tag{5.8}$$

Any solution that meets these two new requirements will also meet the original requirement. The optimal solution is unchanged because y_t does not appear in the goal function and y_t can be squeezed between the two new constraints.

Applying this trick repeatedly, 5.6 can be broken down into,

$$\ln LHS \leq y_t + y_s \qquad y_t + y_s - \ln LHS \geq 0 \tag{5.9}$$
$$y_t \leq \ln z_t \qquad z_t \geq e^{y_t} \tag{5.10}$$
$$z_t \leq -a_t \qquad -a_t - z_t \geq 0 \tag{5.11}$$
$$-a_t \leq 1 - e^{b_t} \qquad a_t \geq e^{b_t} - 1 \tag{5.12}$$
$$b_t \geq -x_t w_m \qquad x_t w_m + b_t \geq 0 \tag{5.13}$$

The left-hand column contains the inequalities needed to prove that the original equation 5.6 is met. The right-hand column contains those same inequalities rewritten in a form supported by the optimizer.

For every intersection contraint specified in the network-wide schema, the bubble balancer creates a constraint like 5.9. For every bubble type, it creates a chain of temporary variables and constraints to get from y_t to x_t (5.10-5.13).

As a practical matter, bubble sizes cannot be fractional, yet convex optimization runs over the real numbers. We thus add a boundary constraint requiring that bubble sizes are ≥ 1 and round fractional results up. While rounding up isn't as accurate as solving the discrete optimization problem[2], the solutions are good enough. However, this makes the $x_t \geq 1$ boundary constaint very important to prevent byzantine corner cases. If there are only 1000 peers, it doesn't make sense for one bubble size to be 1e6 and the other 1e-3. Best would be 1000 to 1, a result which the optimizer will find since it doesn't explore below 1.

5.2 Stability and Uniqueness

Every peer in BubbleStorm runs the optimizer when it obtains new measurement values. These values will be slightly in error and differ between peers. In order for rendezvous to work correctly, all peers must (roughly) agree on the bubble sizes. Thus, we need the optimizer to return the same solution on every peer. Ensuring this consists of two related problems:

- Is there only one optimal solution to the bubble balance problem?
- Do similar measurement values give similar solutions?

The first question hinges upon uniqueness. To guarantee a unique optimal solution, it is sufficient to prove that all the constraints are strictly convex. The intermediate variables we introduced in Section 5.1 will certainly not have unique results; if an optimal solution does not lie on the boundary of a particular intersection constraint, those intermediate variables may lie anywhere between the two loose bounds. However, we don't use these helper variables in the system, so the real question is if the x_t solution is unique. Thus, we need to ask whether the original constraints 5.6 are strictly convex.

Strict convexity is preserved by addition, so it suffices to prove that $f(x_t) := -\ln(1 - e^{-w_m x_t})$ is strictly convex in x_t for $x_t, w_m > 0$. The second derivative,

$$f''(x_t) = \frac{w_m^2 e^{-w_m x_t}}{(1 - e^{-w_m x_t})^2} > 0$$

Thus, the solution to the bubble balance optimization problem is unique.

The question of numerical stability is much harder to answer. Small changes in the input might cause large changes in the output if the problem itself is ill-conditioned or the optimization algorithm misbehaves. We have good reason to believe that our optimization problem is well conditioned. Recall from Section 2.5 that the optimal solution in the limit is simply $\sqrt{\lambda \frac{S_D}{S_Q}}$. This is only ill-conditioned when the traffic S_Q approaches 0, the case we already excluded with the ≥ 1 size constraint. We do not attempt to prove this intuition correct. Instead, we will measure the results over a reasonable range of inputs in Section 5.4.

[2] Discrete convex optimization is NP-hard.

5.3 Optimizer

BubbleStorm includes an implementation of a very simple convex optimizer, based on the algorithms presented in Boyd and Vandenberghe's excellent introductory text [16]. While we could have used an existing convex optimization package, like LOQO [86] or MOSEK [59], these libraries are generally bulky, complex, and cannot be easily embedded into the final application. Furthermore, due to their complexity, their reliable behaviour in an autonomous system is hard to guarantee. Since our problem was relatively simple, we opted to write a small custom convex optimizer that is embedded in the BubbleStorm library.

Our implementation follows classic interior-point barrier techniques [16] quite closely. Imagine the floor of skatepark for a simple two-bubble problem. At each point (x, y) on the floor, the height is set to the bandwidth cost of using bubble sizes x and y. Since the goal function is linear, the floor is thus an inclined plane. Some points in the skate part represent an invalid solution to the balance problem; these points do not achieve the desired rendezvous probability. To prevent these invalid points, we build a half-pipe wall along the constraint.

To find the unique optimal solution, we conceptually let a ball roll to the lowest point in the skate park. This is implemented using Newton's method, which, unlike a normal ball, makes large jumps from point to point. Calculating each successive step of the ball for the general $|T|$-bubble-type problem with all of the attendant intermediate variables requires some higher-dimensional calculus and algebra. To perform these calculations, BubbleStorm includes basic matrix operations to solve the resulting system of equations.

Since a half-pipe has a quite gentle curve, the ball does not quite reach the optimal point. If the half-pipe had been a vertical wall, the ball could have come much closer to the boundary. It is not possible to use a perfectly vertical wall, because this would not represent a continuous (differentiable) function. Furthermore, a very steep wall makes the calculation numerically unstable. For these reason, we use a second, outer loop which steadily increases the steepness of the half-pipe. Eventually, the half-pipe is so steep that the ball comes to rest very close to the optimal solution.

For those who have read Boyd [16], or otherwise have a background in convex optimization, here is a quick design-decision summary of our implementation:

- Logarithmic barrier function added to goal function
- Two-loop interior-point based approach
  - Outer loop increases slope of the barrier
  - Inner loop finds the solution on the central path
- Newton's method optimizes the goal function
  - Exact Hessian matrix used for all supported boundary constraints
  - No support for linear equality constraints
  - Backtracking line search guards against over-shooting boundaries
- Relative-error bound determines termination
  - Estimated error-bound found from the dual problem
- Cholesky factorization used for Hessian matrix inversion
  - Matrices are stored in banded format
  - Cuthill-McKee algorithm used to find band permutation
- The library supports automatically finding an initial interior point value using a "Phase I Method"; however, for bubble balance, a custom initial value is used
  - The outer product of 5.6 is met by the square-root of the worst LHS
  - Each constructed variable derived by solving inequalities backwards
  - A factor of 2 for each inequality is included to keep the initial value away from the barrier function

Population	Speed	Degree
1%	100 MBit	1600
2%	20 MBit	320
7%	10 MBit	160
30%	2 MBit	32
60%	1 MBit	16

Figure 5.1.: Bandwidth distribution in heterogeneous tests

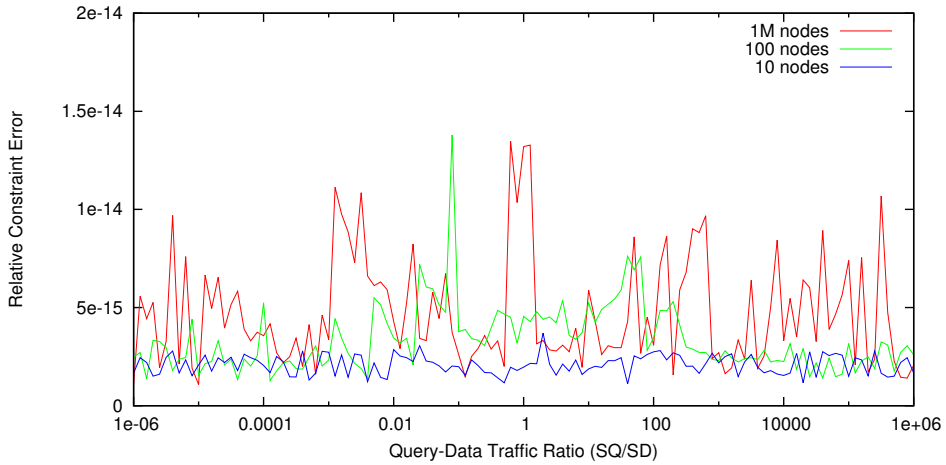


Figure 5.2.: Relative constraint error in a Homogeneous network; below 1e-8.

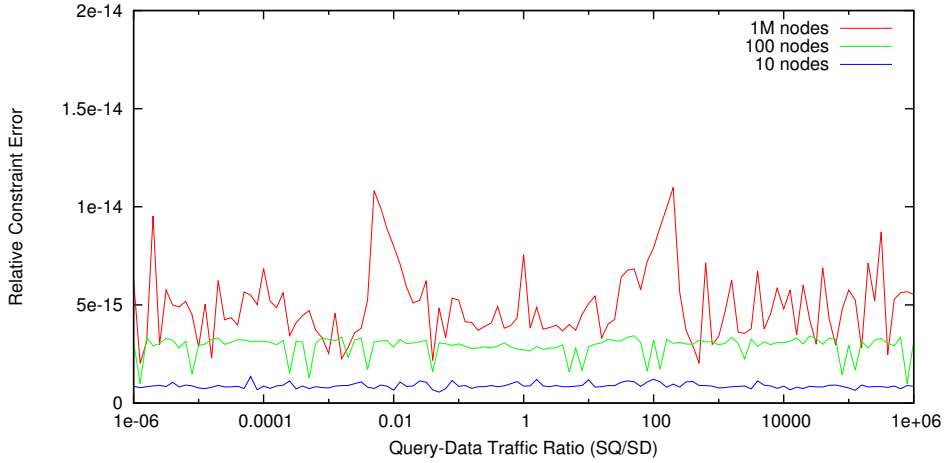


Figure 5.3.: Relative constraint error in a Heterogeneous network; below 1e-8.

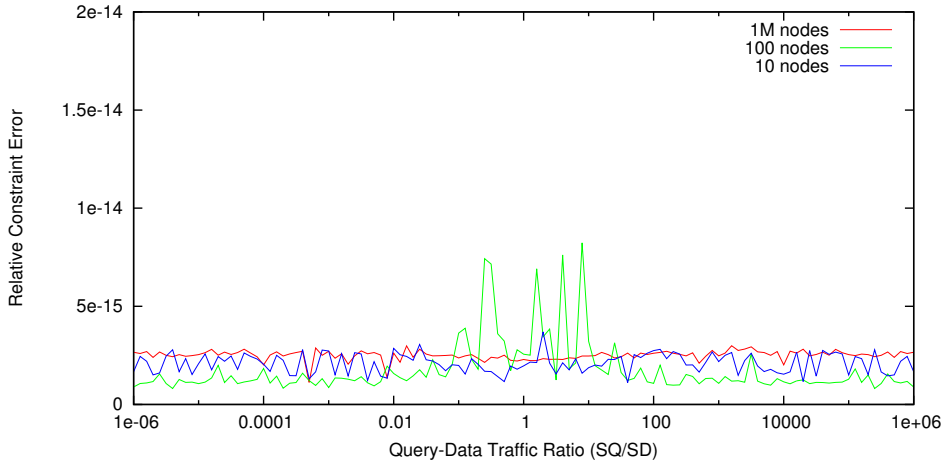


Figure 5.4.: Relative constraint error in a Centralized network; below 1e-8.

5.4 Evaluation

This Section examines the relationship of bubble sizes in BubbleStorm. All figures come from the convex optimizer used to solve the bubble balance equation with $\lambda = 1$. The figures only appear to be smooth as they contain 121 data points per curve. Although the balancer can solve balance between multiple inter-related bubbles, this Section only examines the classic query-data case.

We will examine three network scenarios: homogeneous, heterogeneous, and centralized. In the homogeneous network, every peer has the minimum degree of 16. In the heterogeneous network, the population is broken down according to Figure 5.1. The centralized network has peers of degree 16, except for one peer which has degree equal to the number of peers in the network. All networks, even the centralized network, use normal BubbleStorm topology. That means that although the centralized peer *could* be connected to every other peer, it is expected to be only connected to $1 - e^{-1}$ of them.

Before diving into the results of the optimizer, let's first gain some confidence in its results. While we can't easily test the result directly (the reason we use a convex optimizer is because the solution has no simple analytic form), we can see how closely the optimizer approached the boundary constraint. Figures 5.2-5.4 plot the error in the result. Since we configured the optimizer to approach IEEE double accuracy ($2^{-53} \approx$ 1e-16), the results are pretty good. It is a bit worrisome that the error seems to grow with network size (from 2e-15 average to 4e-15), a phenomenon we have not deeply analyzed. Bubble balance better than 1e-8 is perfectly acceptable for our work.

The first question we probably want to answer is: what does the solution look like? Figures 5.5-5.7 give the answer. The first thing to notice in Figure 5.5 is that query bubble size decreases as it's relative traffic contribution increases. The next thing to notice is that as the network grows, the curve becomes increasing straight—a better fit to $\sqrt{\ }$. Recall from Section 2.5 that BubbleStorm tends towards bubble sizes $\sqrt{S_D/S_Q}$, which this plot clearly demonstrates.

The curves bend away from the ideal straight line because the constraint terms $1 - e^{-w_m x}$ are only a good approximation to $w_m x$ for small $w_m x$. As x grows larger, each additional replica contributes less. This is intuitively explained as follows. Initially, when you place x replicas, you place them on x distinct peers. As you place more, you start to see collisions. Once you've saturated the network, you need to add many replicas to hit one of the few remaining unreached peers. Thus, it becomes unprofitable for the balancer to heavily unbalance the solution. Bubbles larger than (or nearly as large as) the network waste traffic. Thus, we see the curve bend away earlier in smaller networks.

The heterogeneous curves in Figure 5.6 look quite similar to the homogeneous scenario. However, the bubble sizes of the curves are smaller. This is due to the phenomenon already explained in Section 2.5; bigger peers need fewer messages to do the same work as comparatively many smaller peers. We'll come back to this later. The other thing to notice is that the curves never quite reach 1. This stems from the approximation we made to cope with heterogeneity (Lemma 9). By moving w_m into the exponential product, we've capped our ability to balance bubbles a bit early. However, this only

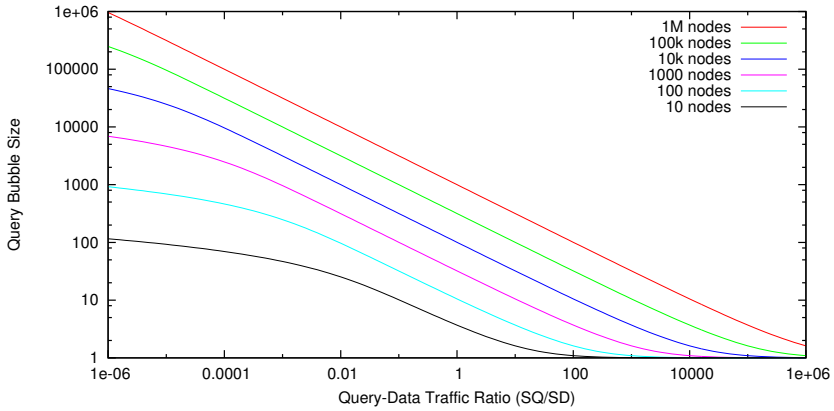


Figure 5.5.: Bubble Size as a function of traffic balance in a Homogeneous network. Bigger networks closely approximate the square-root.

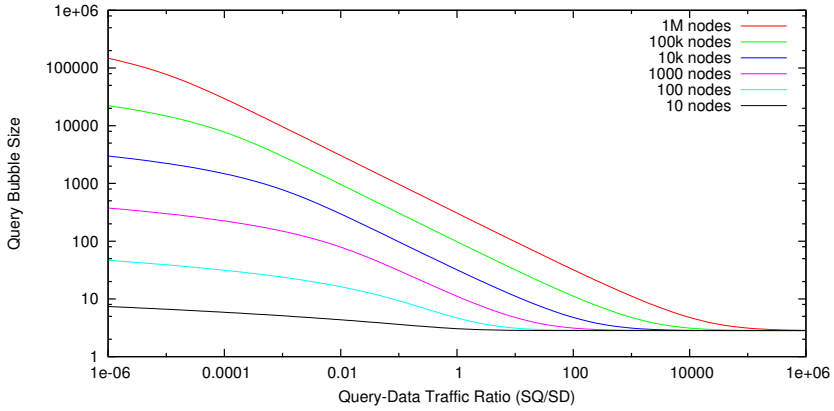


Figure 5.6.: Bubble Size as a function of traffic balance in a Heterogeneous network. Leveraging heterogeneity reduces required bubble sizes.

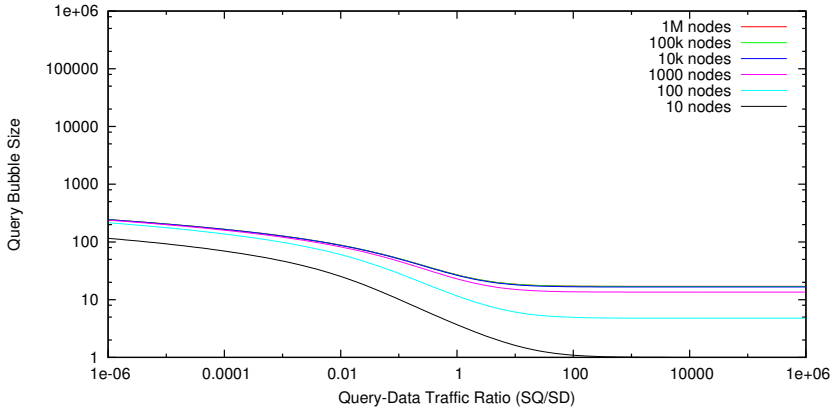


Figure 5.7.: Bubble Size as a function of traffic balance in a Centralized network. A single giant peer causes the system to centralize all traffic.

affects very unbalanced bubbles, where the bottleneck expectation metric (Section 2.1) does not grow with network size[3], and we can thus afford some small constant wastage.

Things are very different in the centralized scenario, Figure 5.7. First, recall that BubbleStorm does *not* guarantee minimal traffic in this case. The assumption that one peer does not dominate the network has been violated. BubbleStorm *does* still guarantee correctness, even in this very degenerate network topology. The effect of a giant peer is quite pronounced. Initially, the 10 peer network doesn't look much different from the homogeneous case. That's obviously because the central peer is not larger than the others in this case. As the network grows, the curve doesn't change that much, because a small bubble will still reach the central node with high probability. The only "balance" possible in this network is related to $\lambda = 1$. One bubble is made large enough to ensure it almost certainly reaches the central peer. The other bubble is made just large enough to have a $1 - e^{-\lambda}$ chance to hit the central peer. As one might expect, that takes just bubble size 16 (1/16 edges lead to the big peer); this is why the all balanced curves flatten out to 16 on the right-side of the graph. Essentially, the bubble balancer has tuned the system to focus on leveraging the centralized server.

The next big question to answer is: how costly are these solutions? The combined traffic cost $xS_D + yS_Q$, is plotted in Figures 5.8-5.10. In our traffic plots, we consider a system where $S_D + S_Q = 1$. Thus, one can interpret the resulting graphs as an amplification factor. If you put 1000 bytes of traffic in (split according to the balance ratio), the y-axis shows how much larger the resulting replica traffic will be.

Certainly, as the network grows, the bubble sizes grow, to the tune of $\sqrt{n}$. The most expensive situation is when the query and data have the same injection rate. When they are unbalanced, the optimizer makes the infrequently used bubble larger. However, as already explained, one cannot push this trade-off much further than the network size. Thus, the traffic flattens out in the face of extreme imbalance.

While the traffic grows as $\sqrt{n}$, it is important to remember that the traffic is split uniformly amongst the peers. Thus, adding more processing peers *decreases* the traffic that each peer will see. On the other hand, adding more peers may bring a corresponding increase to the injected load. In that case, Figure 5.8 faithfully plots the load as the system grows. Please keep in mind that this traffic solution is fundamentally the best one can do. Rendezvous theory dictates the problem requires this many replicas to solve.

As before, the heterogeneous network is cheaper to run than the homogeneous one. Less replicas are needed to ensure rendezvous. Again, approximation Lemma 9 clips the potential imbalance of the heterogeneous solution, preventing it from reaching 1. The centralized network in Figure 5.10 shows the advantage of having a giant peer: traffic

[3] Consider two bubble types $s, t \in T$. If the traffic for s (S_s) is much larger than for t, the bubble balancer will make x_s very small. The balancer cannot make bubbles smaller than size 1. Thus, in such an unbalanced situation, the traffic for s dominates the traffic for t even after balancing ($x_s = 1$). In this situation, bubble type t is flooded through the network ($x_t = n$). As the network grows, the aggregate cost of replicating t grows quadratically ($S_t x_t \in \Theta(n^2)$) and eventually catches up to the dominating cost of s. At this point, the optimal bubble balance is no longer at $x_s < 1$ and the traffic of s and t are balanced to parity. Before this point, however, aggregate traffic is only growing linearly (t's quadratic growth is dominated by s's linear growth), and thus the bottleneck utilization does not grow with n.

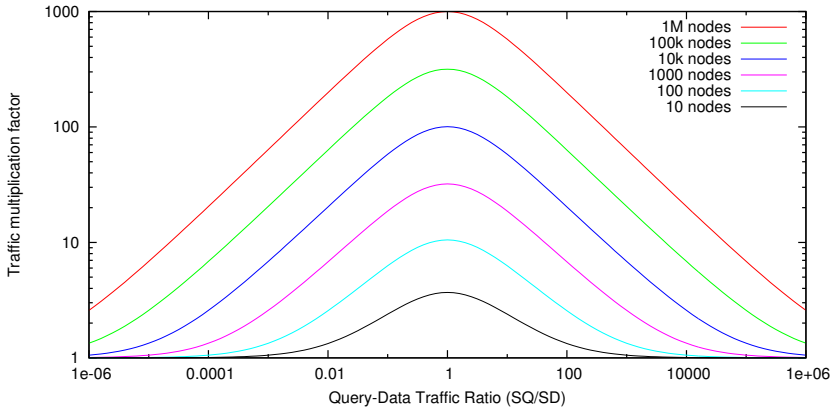


Figure 5.8.: Traffic as a function of balance in a Homogeneous network. With balanced traffic, bandwidth grows as $\sqrt{n}$. Unbalanced traffic equals big savings.

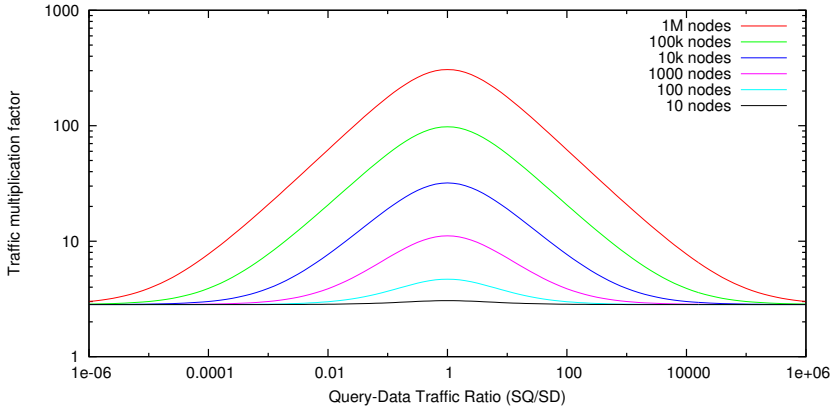


Figure 5.9.: Traffic as a function of balance in a Heterogeneous network. Leveraging big peers for big savings. The w_m approximation clips the lower limit.

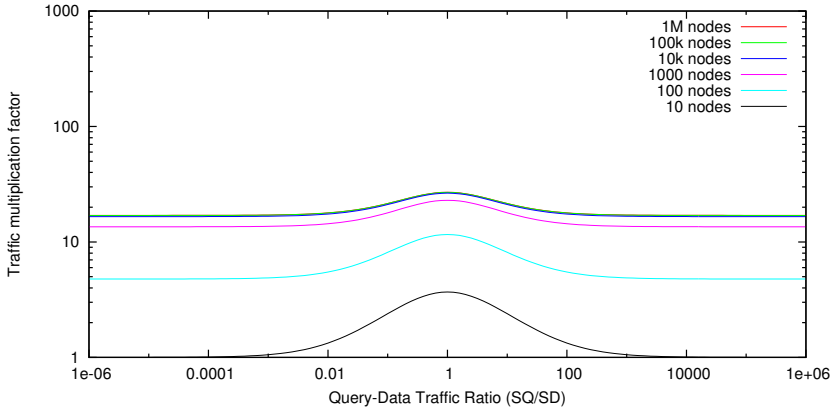


Figure 5.10.: Traffic as a function of balance in a Centralized network. A single big peer completely halts bandwidth growth; Cloud services work.

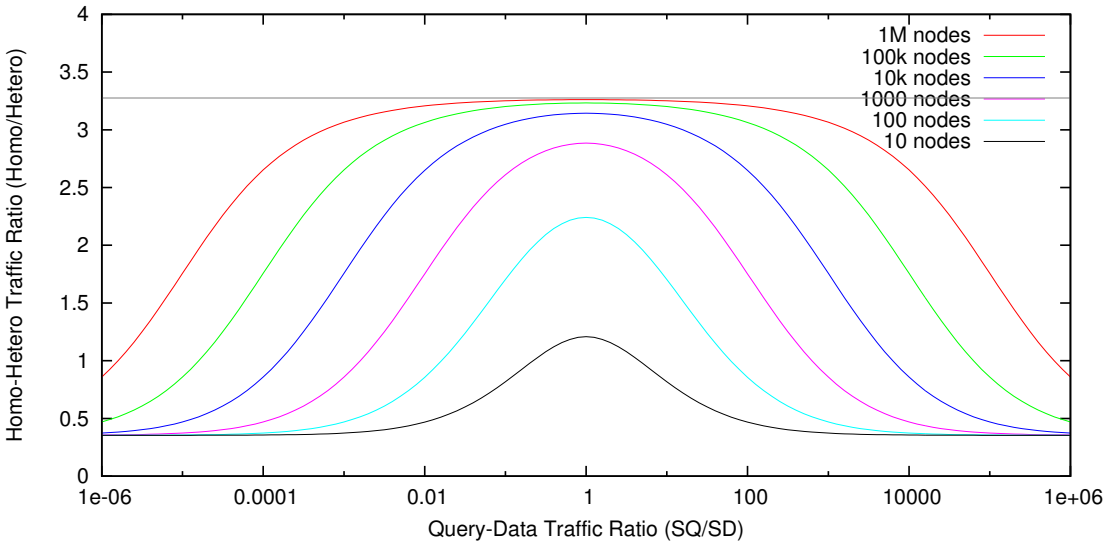


Figure 5.11.: Traffic improvement from heterogeneous networking. Gains in balanced scenarios exactly equal the prediction. As imbalance increases, optimal balance can no longer be attained and ultimatedly the w_m approximation penalizes the heterogeneous network (where the cost is so cheap it hardly matters).

costs do not grow with network size. Of course, it may be infeasible to have a peer this powerful, necessitating a peer-to-peer solution.

To explore the gain from heterogeneity more closely, Figure 5.11 plots the ratio of homogeneous traffic to heterogeneous. As we can see, for the critical scenario where bubbles are equal sized, there is an improvement of almost factor 2. In truth, the heterogeneous system should *always* be an improvement over the homogeneous system. However, the approximation we used to simplify automatic bubble balance clips heterogeneous imbalance. Thus, as the imbalance grows, the homogeneous solution can carry the trade-off further than the heterogeneous solution, resulting in a situation where BubbleStorm would be better off treating all peers as though they had equal capacity.

The gain from heterogeneity can be readily calculated. In the limit, $xy \to D_1^2/D_2$. For a homogeneous network $D_1^2/D_2 = n$. For our heterogeneous network it is $\approx n/10.73$. Splitting the gain between the two bubbles, $\sqrt{10.73} \approx 3.27$ which is what we see in Figure 5.11.

6 Topology Theory

In peer-to-peer networks, peers do not know the addresses of all other peers. Instead, one peer knows the address of a small subset of the peers in the network. These peers are called its neighbours. Considered as a whole, the graph of neighbour relationships is called the network topology.

Designing a good network topology is the focus of much research. Since every system has its own requirements, there are at least as many proposed topologies as real peer-to-peer systems. This chapter examines BubbleStorm's specific requirements, covers elementary random graph theory, and outlines the principles behind our topology.

BubbleStorm uses a random graph network topology. There are many reasons for this choice, but the most important stems from rendezvous theory. We want to select peers at random to place replicas on them. If every neighbour connection is the result of a random process, then selecting a random peer is naïvely as easy as following an edge.

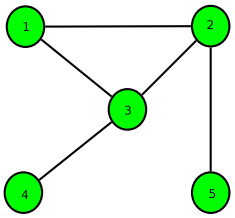
This high-level idea plays out to our advantage in a number of ways. In a small area, a random network topology looks a lot like a tree [14]. Thus, peers can replicate queries to their neighbours without being overly concerned that too many peers will see a query twice. The second eigenvalue of a random graph (explained in the next section) is also small. This ensures that short random walks select a random peer independently from the current topology. Furthermore, it speeds up the mixing time required by our measurement protocol (Chapter 8).

Random graphs are also especially attractive in a peer-to-peer setting. Even when a large fraction of the edges are removed, they stay connected [22]. Using a random graph also gives us a lot of flexibility to deal with the Internet's incomplete connectivity. If two peers cannot directly communicate (a fairly common occurrence), we can roll dice again and connect two different peers[1]. Finally, constructing a random graph does not require global knowledge.

6.1 Random walks and expansion

Before diving into random graph models, let's first review some basic graph theory about random walks and expansion. A random walk moves from peer to peer by selecting a neighbour uniformly at random. Let G denote the adjacency matrix of the network's graph and V a diagonal matrix with the vertex degrees; see Figure 6.1 for an example. Then GV^{-1} is the Stochastic matrix which describes the transition probabilities used to select the next peer in a random walk.

[1] Contrast this with the topology in structured systems like DHTs, where certain connections between peers are required by the relationship of their keys. If two peers cannot communicate, the system correctness is compromised.



$$G = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}, V = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 6.1.: Adjacency and vertex degree matrixes for an example graph.

Define w to be the vector where $w_u := \deg(u)/\sum_v \deg(v)$. Direct calculation shows that $GV^{-1}w = w$ and therefore w is an eigenvector with eigenvalue 1.

Assuming that the network is connected, it is impossible to permute G into upper-triangular form,

$$P(GV^{-1})P^{-1} \neq \begin{pmatrix} A & B \\ 0 & C \end{pmatrix}, \text{for all } P,A,B,C$$

as this would demonstrate a cut of the graph. By the Perron-Frobenius Theorem for irreducible matrices, it follows that any eigenvector with all positive entries must correspond to the largest magnitude eigenvalue of the matrix. Since w has positive entries, the largest eigenvalue is thus 1.

Furthermore, since the graph is symmetric, every edge forms a 2-cycle. The existence of even one odd cycle immediately implies that the graph is aperiodic. For random graphs, this is almost surely the case [14]. As a consequence, every other eigenvalue from the Perron-Frobenius Theorem must have magnitude < 1.

Since $GV^{-1} = V^{0.5}(V^{-0.5}GV^{-0.5})V^{-0.5}$, which is a similarity transform, $V^{-0.5}GV^{-0.5}$ has the same eigenvalues as GV^{-1}. Furthermore, as G is a symmetric matrix, so too is $V^{-0.5}GV^{-0.5}$. By the Spectral Theorem, we can conclude that $V^{-0.5}GV^{-0.5}$ has real eigenvalues corresponding to an orthonormal basis of real eigenvectors. Let λ_i and v_i denote them respectively, so that $V^{-0.5}GV^{-0.5} = \sum_i \lambda_i v_i v_i^T$. Notice that $v_1 \propto V^{-0.5}w$ and $V^{-0.5}v_1 \propto \mathbf{1}$.

Now we are positioned to analyze how a random walk behaves. Suppose we have an initial probability distribution x for first step of the walk. As an example, if the walk begins on peer u, then x is 0 at all entries except the u^{th} where it is 1. In any case, the sum of all elements $\sum_i x_i = \mathbf{1}^T x = 1$.

Thus the result of an m-step random walk is w plus a remainder term,

$$\begin{aligned}
(GV^{-1})^m x &= V^{0.5}(V^{-0.5}GV^{-0.5})^m V^{-0.5}x \\
&= V^{0.5}(\sum_i \lambda_i v_i v_i^T)^m V^{-0.5}x \\
&= V^{0.5}(\sum_i \lambda_i^m v_i v_i^T)V^{-0.5}x \\
&= \sum_i \lambda_i^m (V^{0.5}v_i)(V^{-0.5}v_i)^T x \\
&= 1w\mathbf{1}^T x + \sum_{i>1} \lambda_i^m (V^{0.5}v_i)(V^{-0.5}v_i)^T x \\
&= w + V^{0.5}(\sum_{i>1} \lambda_i^m v_i v_i^T)V^{-0.5}x
\end{aligned}$$

.

To bound the remainder term, take the matrix norms of the components,

$$\begin{aligned}
|(GV^{-1})^m x - w| &= \left| V^{0.5}(\sum_{i>1} \lambda_i^m v_i v_i^T)V^{-0.5}x \right| \\
&\leq |V^{0.5}| \left| \sum_{i>1} \lambda_i^m v_i v_i^T \right| |V^{-0.5}||x| \\
&= \sqrt{\max_u \deg(u)}\lambda_2^m \sqrt{1/\min_u \deg(u)}1 \\
&= \lambda_2^m \sqrt{\frac{\max_u \deg(u)}{\min_u \deg(u)}}
\end{aligned}$$

Thus, the second largest eigenvalue λ_2 clearly plays a critical role in how quickly the system converges to w. Furthermore, the steady state w has probability proportional to degree. This suggests an easy way to distribute a rendezvous workload over a graph with heterogeneous peers; set every peer's network degree proportional to its capacity.

The second eigenvalue is already of immediate interest to us for random walks (used to build the topology), our mixing-based measurement protocol, and replica placement using bubblecast. What's more, it is deeply related to the concept of edge expansion. The edge expansion of a graph is defined as,

$$h(G) := \min_{S \subset U : 0 < |S| < |U|/2} \frac{|\partial(S)|}{|S|}$$

where $\partial(S)$ is the set of edges with exactly one end-point in S.

The edge expansion is interesting, because useful communication networks tend to have high expansion. High expansion guarantees a short network diameter and the ability to reach an exponentially growing number of peers per hop. A high expansion also makes it very hard to cut the network; the linear increase in connections leading

out of small subsets of peers causes the probability to disconnect from the core network to fall exponentially. Thus, when a large number of network connections are cut, an expander is very likely to remain connected. If, however, some peers do get disconnected, they will nearly always be in very small and easily recognizable islands. In practice, this provides an implementation with an easy hint that disconnected peers must seek further connections to reestablish connectivity.

The Cheeger inequality [62] relates the edge expansion to the second eigenvalue. If a network has regular degree d, it states,

$$\frac{1}{2}(1-|\lambda_2|) \leq \frac{h(G)}{d} \leq \sqrt{2(1-|\lambda_2|)}$$

Recall that we took the eigenvalues of the transition probability matrix GV^{-1}, not G as used by [62] and [29]. Thus, our $|\lambda_2|$ replaces $|\lambda_2/d|$ in their notation.

At this point it should be painfully obvious that the second eigenvalue is the single most important property of a peer-to-peer network topology. Fortunately, Joel Friedman proved Alon's second eigenvalue conjecture [29], for a random graph model very similar to the one we use in BubbleStorm[2]. That result states,

$$|\lambda_2| < 2\frac{\sqrt{d-1}}{d}$$

While this result has only been proven for homogeneous random graphs, it seems quite safe to presume that increasing the degree of some vertices will only serve to decrease the second eigenvalue further.

6.2 BubbleStorm Topological Model

The BubbleStorm topology is a degree constrained random graph. Every peer selects an (even) *desired* degree. This degree should be chosen proportionally to the peer's capacity; $\deg(u) \propto S_u$. The lowest capacity peers in the BubbleStorm implementation are defined to have degree=16. The topology protocol (Chapter 7) will ensure that a peer's degree stays quite close to its desired degree.

A well known property of graphs with even degree nodes is that they have Euler tours. An Euler tour traverses every edge in the graph exactly once. For example, Figure 6.2 shows a graph and a corresponding Euler tour. Notice that the Euler tour is a cyclic permutation of the vertexes, where a vertex v appears $\deg(v)/2$ times in the permutation. We call each place it appears in the permutation a *location*.

Our random graph construction assigns equal probability to every permutation. Every vertex (peer) v in the graph chooses its own even degree $d = \deg(v)$. The model then writes down a list of location symbols. Each vertex is listed with symbol v_i for $i \in [0, d/2)$. For example, if peers a, b, and c all want degree 6, then the symbols $a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2$ are included in the list of symbols. Every permutation of

[2] In fact, we intentionally designed the BubbleStorm topology protocol to implement a graph as similar to Friedman's theorem as was practical.

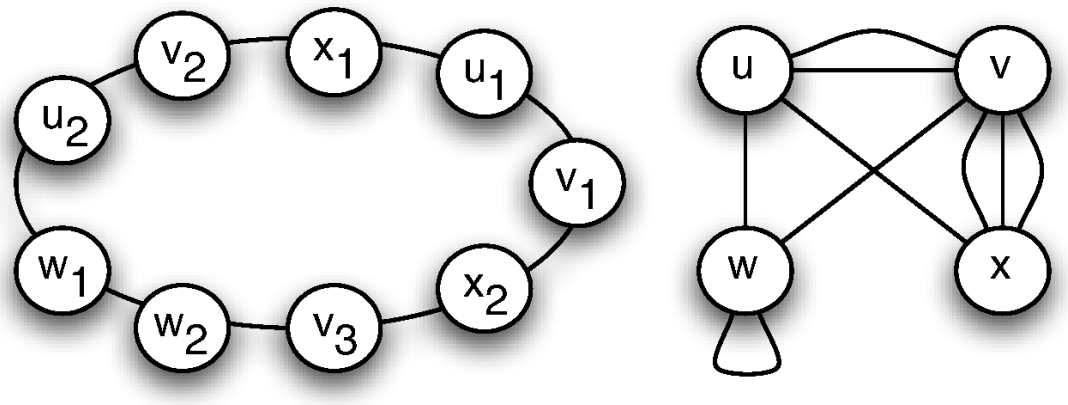


Figure 6.2.: An Euler tour of an example graph

these symbols is assigned equal probability, and the corresponding graph is drawn. Given a particular graph, its probability in this model is the sum of the probabilities of each matching permutation.

There are several reasons this model works well. First, it allows fine-grained control over the load each peer receives by tuning the degree / number of locations. Second, it is easy to add and remove nodes from a random permutation; just pick a random position in the ring and insert a location there. Finally, it makes random sampling for rendezvous replica placement easy. The first explored edge has (by definition) a w_u chance of reaching peer u as required by Theorem 6. Each subsequently explored edge has a slightly higher chance of reaching new peers. This tends to make the network perform slightly better than simple uniform sampling. We can ignore this good news without impacting correctness.

The only problem is that whenever you reach a given peer twice, you are quite likely to subsequently rediscover peers you have already seen. This *collision* chance and the resultant sampling dependency exists in every network topology that is a strict subset of the complete graph. Therefore, it is an unavoidable problem for a peer-to-peer network. Fortunately, this problem is mitigated by two factors. First, given our choice of minimum degree 16, the bubblecast protocol (which places the replicas; see Chapter 9) expects a collision to result in < 1 chained collisions. Therefore, when a collision does happen, it doesn't cascade too far. Second, collisions above those you'd expect from the Birthday paradox are relatively rare in random graphs.

Unfortunately, the proof of this last claim is phrased in terms of perfect matchings, a slightly different model of random graphs used by Bollobás [14]. But fortunately, our model contains that model as a subgraph. By proving this relationship, we can claim a number of useful things about our topology.

Under the perfect matching model, one imagines each peer to have one dot per desired degree. Naturally, there must be an even number of total dots for an undirected graph to be possible. Then, we randomly pair up these dots (a perfect matching) and

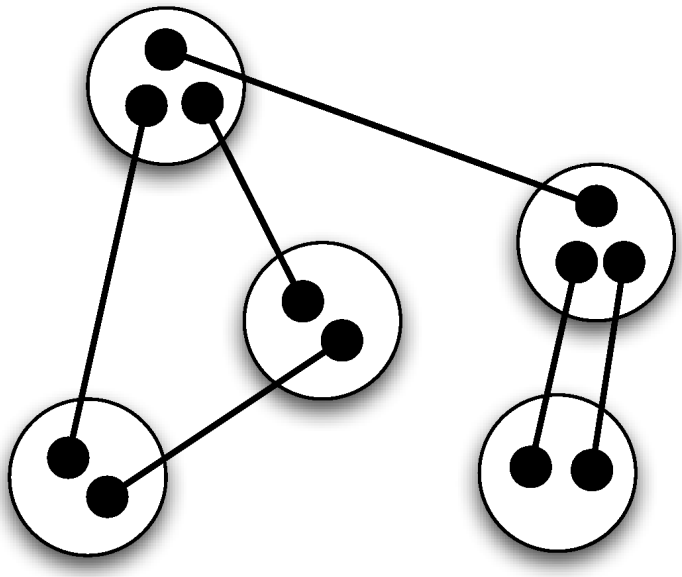


Figure 6.3.: An example graph constructed using a perfect matching

connect them. For every two connected dots, add an edge between the two peers from which they came. See Figure 6.3 for an example.

One technical detail is that Bollobás [14] rejects any matching that would result in double edges or self loops. BubbleStorm allows double edges and self loops by design. Fortunately, most proofs in [14] actually operate on the perfect matching model without the rejection step. Only in the last stage of these proofs is the result converted to a model which rejects double edges and self loops.

To see how the BubbleStorm model includes the perfect matching model, consider the BubbleStorm locations. Pair up the locations as a perfect matching. Now roll some extra dice to label one of those locations to come first and then randomly order the pairs to build a completed permutation. Figure 6.4 shows how the new dashed red edges are added to the original perfect matching to obtain the containing BubbleStorm topology. This permutation includes all the edges the perfect matching model would have.

Now we can cite Bollobás [14] and Chung [22] to claim:

- Nearly every small connected subset of the BubbleStorm graph looks like a tree (and thus collisions before the Birthday Paradox are unlikely)
- There is on average exactly one cycle of each length (up to $O(\sqrt{n})$)
- The network is almost surely connected (despite broken edges in the next section)
- Up to 83% of the edges can be cut and the system will retain a giant-component (containing a fraction of peers related to the failure percentage)
- Vertexes disconnected from the giant component will be in small islands

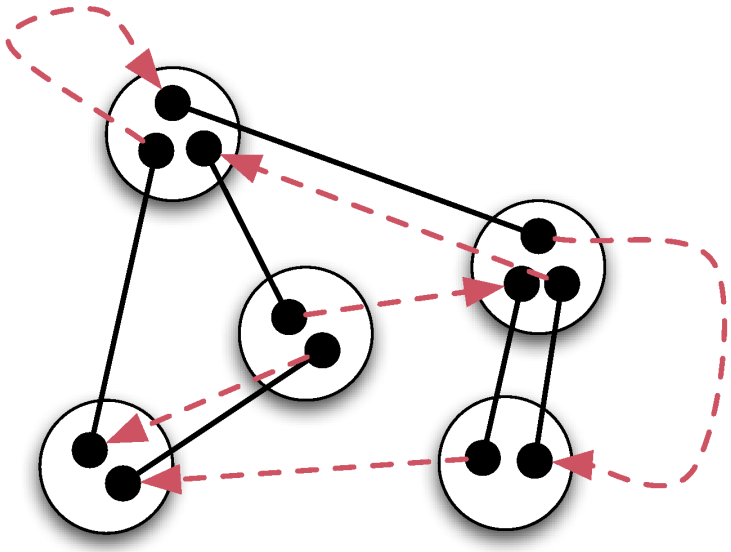


Figure 6.4.: Half of BubbleStorm edges correspond to a perfect matching

6.3 Broken Edges

In a practical system, it is unavoidable that some peers will crash and some pairs will be unable to communicate. Both of these effects can lead to broken edges in the cyclic permutation. Fortunately, this can be easily worked around.

When a peer u leaves, it should normally coordinate the integrity of the Eulerian cycle. For each location, it connects the two peers on either side of it in the cycle. In this way, u is removed from the cycle, which remains intact.

If, however, peer u crashes, it cannot tell its neighbours to connect to each other. Thus, these two peers have their degree reduced by one and the Eulerian cycle is severed. Peers could gossip about neighbours "near" themselves in the cycle (like Chord [75] or Pastry [70]). Then they would be able to reconnect when just a few peers crash. However, this is unnecessary and would not solve the next problem.

In the Internet at large, some peers cannot communicate with others. This might be a routing problem, a firewall configuration issue, or a packet congestion condition. Regardless of the cause, some of the edges selected for construction by the random permutation cannot be formed.

BubbleStorm does not attempt to repair broken edges. When an edge breaks, it stays broken. However, if a location has two broken edges, it is removed (since this does not affect the degree). As peers come and go, the size of an unbroken segment in the permutation constantly fluctuates. Whenever a segment is reduced to one member, it vanishes. Thus, if the system were run long enough (growing/shrinking the cycle segments) without breaking any further edges, eventually only a single unbroken segment would remain. Since this tendency is balanced by freshly created broken edges, the

system has steady-state where a stable fraction of edges are broken. The exact ratio depends on the frequency of crash events to leave events.

Broken edges have two effects. First, the degree of affected peers is reduced. Since peers must maintain a fixed degree to ensure load-balance, this is something BubbleStorm actively corrects (Section 7.1.1). Correcting peer degrees costs some traffic; thus BubbleStorm aims to maintain the Eulerian cycle whenever possible. Remember, peers that leave or join the Eulerian cycle correctly do not affect established peers' degrees. The other problem with broken edges is that the actual graph is a subgraph of the random graph model.

As long as the broken edges are chosen randomly, the perfect matching model can be used to demonstrate that BubbleStorm still contains a random graph as a subgraph. Therefore, it remains connected and the rendezvous continues to work. Crashes happen independently of a peer's physical internet connectivity, so they fall into this category and don't impact system correctness. However, sometimes the broken edges are correlated. For example, if one of the fiber cables which carry the bulk of international traffic gets cut, the edges which get broken will be correlated.

Our experiments (Section 9.4) show that as long as the broken edges do not nearly partition the graph, BubbleStorm works. A true cut of the underlying Internet results in two independent systems running side-by-side. Naturally, queries in one system cannot rendezvous with new data in the other.

6.4 Related Work

There are hundreds of proposed peer-to-peer topologies. Roughly speaking, they fall into two categories: structured and unstructured (Figure 6.5). The exact definition of (un)structured systems depends upon the person speaking, but everyone agrees there is a distinction. Many researchers conflate an arbitrary network topology (like Gnutella [42]) with unstructured networks. However, in the context of this thesis, it seems more useful to instead categorize based on who picks the network's connections. If the system operation requires certain connections between peers, this is what I define as structured. If, instead, correct system operation only requires that the graph has certain features, this is unstructured.

For example, Gnutella [42] has an arbitrary graph topology where peers may connect to any other chosen peer. It only requires graph connectivity to process requests. In contrast, both Chord [75] and Pastry [70] assign peers an identifier and then sort peers by identifier into a ring. Here, two peers with adjacent identifiers must be connected in the topology, or the routing algorithm can reach a dead-end and fail.

Usually, structured systems require connections to facilitate efficient key-based routing. The ring-based systems [70, 75] use their connections to guarantee routability along the ring. Typically, ring-based systems include extra far-reaching edges to support faster routing than walking the entire ring. In this way they are similar to the Tree-based systems, like P-Grid [1] and Kademlia [56], which organize their routing tables based on matching identifier prefixes. De Bruijn systems, like Omega [25] or Koorde [41],

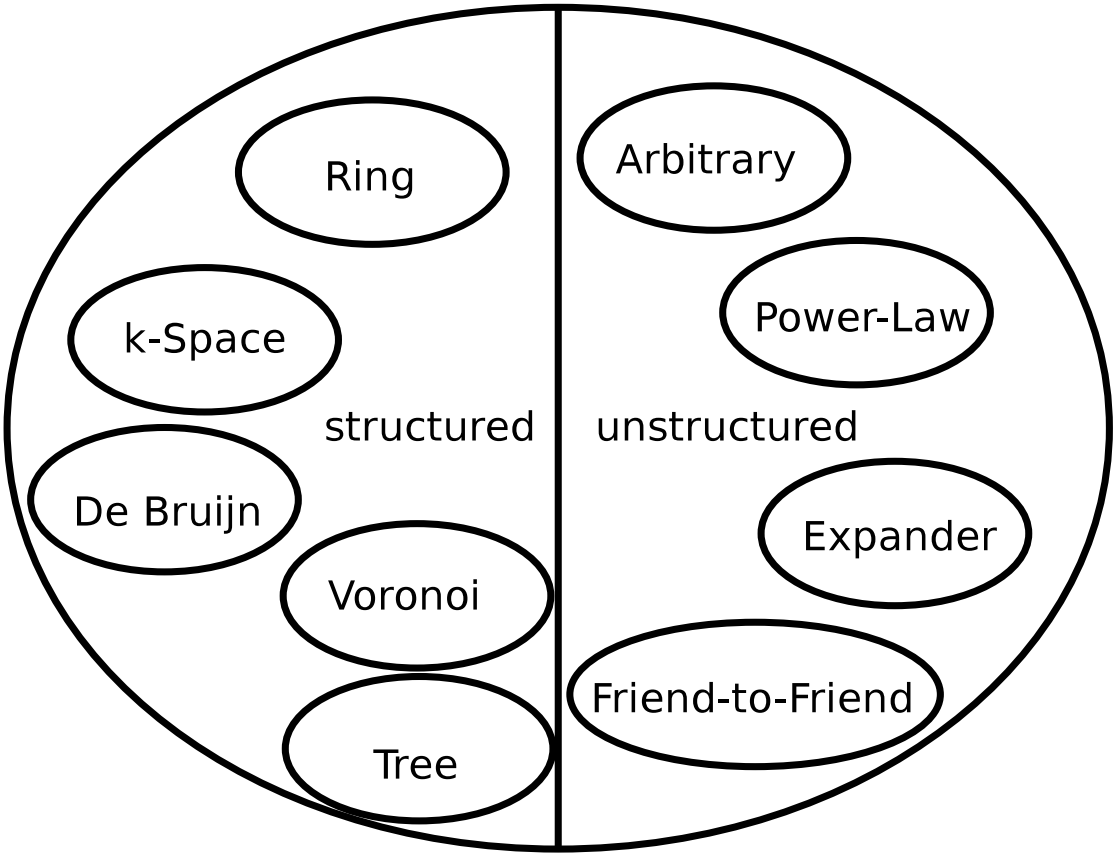


Figure 6.5.: A catalogue of peer-to-peer topologies

model De Bruijn graphs, using bit-shift semantics to quickly reach a given identifier. All of these systems can route to a given identifier in $O(\log n)$ hops.

Some structured systems do not route to a one-dimensional key. The CAN system [68] models a k-dimensional Euclidean space, potentially supporting routing to a k-tuple of keys. Voronoi systems [47, 48] connect peers based on a position identifier. Each peer is responsible for positions closest to it, and it connects to peers responsible for adjacent positions. This is useful in games where players see other nearby players.

This is only a small sampling of structured systems. At the height of peer-to-peer popularity, whenever a graduate student saw a new problem, he could be sure of a few easy publications by inventing a new structure. As long as a structured system remains a research paper, everything looks good. However, in the real-world, the edges these systems require may not be possible! The Internet is plagued with incomplete connectivity. When implemented for use in practice, these systems must work around this problem. For example, ring systems should not only connect to peers with the nearest identifier, but also a few more. Both Chord [75] and Pastry [70] do this, but still a message can reach a dead-end. Their greedy routing protocols move a message closer to the target identifier with each hop. This can get the message stuck on a peer

which is not the target, but is unable to connect to any peer closer to the target. The connections to reach the target exist, but without back-tracking they are useless.

In contrast, unstructured systems do not depend on specific connections. Instead, they focus on graph properties. For example, BubbleStorm, Ferreira et al.'s random walk system [28], and Freenet [23] all rely on graph expansion. No particular edge is required, but the graph must have high expansion. This property is relatively easy to guarantee with high probability; just maintain a random graph. Unlike structured systems, there is no need to work-around troublesome connectivity issues. These systems were designed to work with the incomplete connectivity the Internet actually provides.

Like expander graphs, power-law graphs can also be used to build peer-to-peer systems, see for example Percolation Search [72]. However, power-law graphs have, by their nature, high degree peers which connect most of the network. Power-law graphs generally appear when a joining peer tends to connect to peers with already high degree. This was true of the original Gnutella protocol, which works on arbitrary connected graphs. However, these high degree peers will receive a proportionally high share of traffic (Section 6.1). It's hard to call a system which places most of its load on a few servers a peer-to-peer system. As we have already demonstrated (Chapter 2), these systems do not approach optimal bottleneck utilization. Power-law graphs are interesting where they appear naturally, but seem a poor choice for an intentional topology design.

Of final interest are the friend-to-friend networks; for example, Gnunet [10], Freenet [23], or a survey [21]. These networks form connections between the computers of people who know each other. Due to the small-worlds phenomenon [8, 45], the resulting graph should have good expansion. Thus, in some sense they are another variation on expander systems, but unlike random graph approaches they do not enforce/guarantee expansion. It would be quite interesting to find a variation of BubbleStorm that worked on these networks.

7 Topology Protocol

In principle, all the topology protocol must do is insert peer locations into the Eulerian cycle at random positions on join and remove those locations on leave (compare Figure 7.1 to Figure 6.2). In practice, it is one of the most complicated components in BubbleStorm. Although every layer of BubbleStorm assumes that the underlying layers can fail, the topology layer runs directly atop the Internet. It must deal with potential timeout or failure at every step of operation. Furthermore, it must be programmed as an event-driven state machine. While there are some exchanges of message which occur in sequence, there are many other messages which must be processed in any order. Fortunately, the layers above the topology are much easier to reason about once the topology has smoothed away most of the underlying complexity of the Internet.

At its simplest, the task of the topology protocol is to interconnect the peers and ship messages between them. To improve the performance of the rendezvous system, the topology protocol strives to lose as few messages as possible. Some of our earlier designs occasionally created black-holes where a peer would receive messages that it could not route further (ie: it had degree<3 temporarily during join/leave). To prevent loss, it is also important to cleanly tear a connection down. Thus, we must have two relationships between peers: client-server for half-duplex transmission and peer-to-peer for full-duplex. Finally, the topology is responsible for maintaining degree close to the desired degree for each peer.

The overall architecture is illustrated in Figure 7.2. The degree tuner accepts goals from the user. These goals include JOIN or LEAVE. As events and state changes occur, the degree tuner adjusts the goals of each subordinate location appropriately. If the

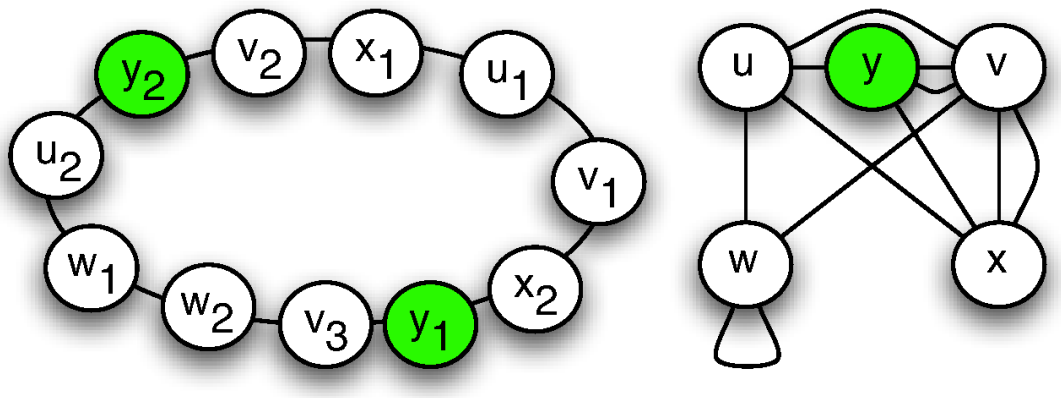


Figure 7.1.: A new peer (green) has joined the network graph

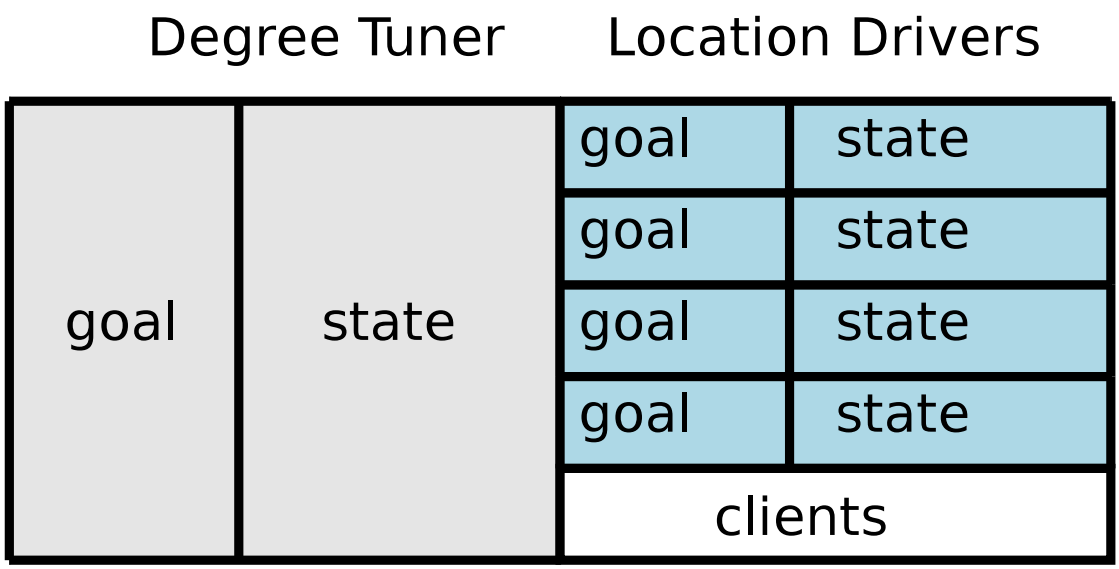


Figure 7.2.: The topology's event-driven state machine components

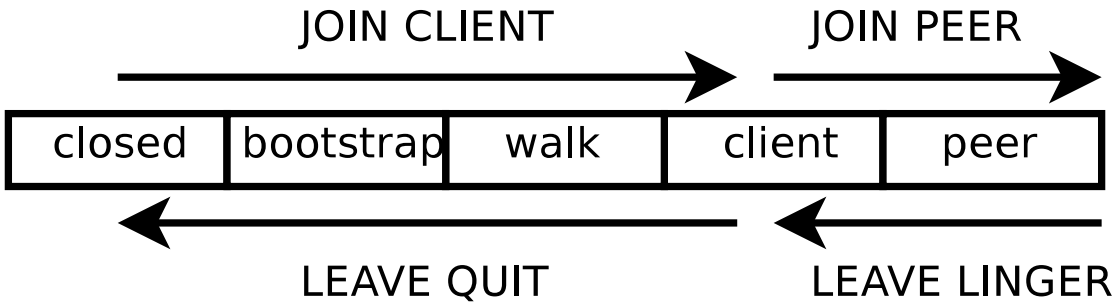


Figure 7.3.: The state transitions a location goes through depending on its goal

topology should join, then the degree tuner sets some of the location goals to join. Each location operates independently of the other locations and seeks to follow Figure 7.3.

7.1 The Ring

As in the BubbleStorm graph model (Section 6.2), each location has a position in the Eulerian cycle. These locations manage two connections, to the successor and predecessor in the cycle. In the implementation, we designate the predecessor as the master. Thus, each location acts as a slave to its predecessor and master to its successor.

A location can act as either a client or a peer. Locations in the client state have a connection to another peer's location which acts as a server. This peer location has been randomly selected from the Eulerian cycle. Later, the location can upgrade to peer status by becoming slave to the server's location and master to the server's old slave. Servers

do not route messages over their clients. Peers and clients are free to route message to their neighbours.

It might seem strange that BubbleStorm allows self-loops and double-edges in the topology. Routing over these edges is certainly wasteful. However, there are three reasons we made this design decision. First, as the network grows, the proportion of these 'wasteful' edges vanishes. So in large networks, there is no performance lost. Second, in small networks there are not enough peers to meet the desired degree of peers. By allowing self-loops and double-edges, we avoid needing special-case code to handle these situations. This also makes it very simple to bootstrap a new network (Section 7.2). Finally, double-edges and self-loops are natural consequences of our permutation graph model. Leaving them unhacked keeps the mathematical analysis and software implementation much simpler.

7.1.1 Degree Tuning

The degree tuner ensures that the peer joins/leaves the network, maintains its desired degree, and doesn't become a routing black hole. To accomplish these feats, it must know the current peer degree.

Each location contributes to the peer's overall degree. When fully connected, it contributes degree 2. If it has a broken edge, it contributes degree 1. If it has fully left, it contributes degree 0. Unfortunately, this contribution cannot be precisely quantified in practice. Whenever there is an in-progress operation, one cannot know what the resulting degree will be. Instead, each location provides upper and lower bounds on its degree based on its current state and goal. If a location is leaving, then it's minimum degree is 0. Otherwise it is the current number of functioning connections. Conversely, the maximum degree assumes that all edges being modified will succeed and contribute to the degree.

Taken as a whole, the degree tuner obtains an upper-bound (max) and lower-bound (min) on the peer degree. This state combined with the tuner's goal dictates how the tuner adjusts the subordinate location goals. A rough outline of these transitions is sketched in Table 7.4. Keep in mind that the min degree is only decreased when an

goal	condition	action
JOIN	$\text{min} < 4$	Set 8 locations to JOIN CLIENT
JOIN	$\text{min} \geq 4$	Set every JOIN CLIENT to JOIN PEER
		Set exactly deg/2 locations to JOIN PEER
		Inform user that the join is complete
JOIN	$\text{min} \geq \text{maxDeg}$	Change one JOIN PEER to LEAVE QUIT
JOIN	$\text{max} \leq \text{minDeg}$	Change one LEAVE QUIT to JOIN PEER
LEAVE	$\text{max} > 0$	Set every JOIN to LEAVE LINGER
LEAVE	$\text{max} = 0$	Set every LEAVE LINGER to LEAVE QUIT
		Inform user that the leave is complete

Figure 7.4.: The degree tuner conditionally executes actions based on its goal.

edge breaks; a correctly leaving peer does not affect the degree of the remaining peers. Similarly, the max degree is only increased when an edge breaks (if the last edge at a location breaks, the location automatically seeks to reconnect—probably obtaining two neighbours instead of one).

Since adding a new location adds two edges, it is not possible to control the degree exactly. Instead, the tuner aims to keep the degree between minDeg and maxDeg. While the most obvious approach might be to set minDeg and maxDeg to ± 1 of the desired degree, there is an easy optimization to be made. Consider a coin with 1 on one side and -1 on the other. If after each flip the total is computed, it takes on average m flips until the total exceeds $\pm\sqrt{m}$ [36]. Since the coin flips in our scenario correspond to edge break events, the higher a peer's degree, the more events it sees. By setting our tolerance to $\pm\sqrt{\deg/16}$, we ensure that peers spend the same work tuning their links.

The degree tuning rules take special care not to switch to being a full peer until the node has at least three servers. This way, when the peer begins receiving routed messages, it already has two alternative paths to route those message over. Indeed, it probably has five if the upgrades go well and many more once the other JOIN PEER locations complete. Similarly, when leaving a peer downgrades all of its connections to client mode. This way any queued traffic it has yet to process can still be routed. These two precautions greatly reduce the chance that a joining/leaving peer becomes a routing blackhole, increasing the robustness of the system.

7.1.2 Location Selection

To find a server (or peer), locations must select a random position in the Eulerian cycle. To achieve this, BubbleStorm uses a biased random walk. We already saw in Section 6.1 that a random walk selects peers proportionally to their degree. However, if there are broken edges, selecting a peer proportional to degree is not the same as selecting a location uniformly at random. This is because half-broken locations do not contribute two edges and are thus under-represented in the peer degree distribution.

To solve this problem, we bias the random walk using a holding probability. At each step in the random walk, the forwarding peer roles a die to select the outgoing edge. However, it also includes phantom broken edges in this die roll. If the die selects a real edge, the message is forwarded with reduced hop count. If the die selects a phantom edge, the peer pretends to have sent the message to itself (still reducing the hop count by one). This holding probability causes the steady state probability to converge to selecting locations uniformly at random.

To see why this works, fill in a new matrix G which includes values h_u in the diagonal that count the number of broken edges at peer u. Direct calculation will verify that w is indeed the eigenvector with eigenvalue=1. G is still symmetric so the calculation done in Section 6.1 still shows that the probability distribution converges to w proportional to λ_2^m after m steps.

Of course, the λ_2 here is for this strange graph with self-loops for broken edges. Nevertheless, in BubbleStorm no more than half the edges can be broken thanks to the policy that a fully broken location finds a new position in the cycle. Thus, whatever

would work for the random graph will work for this graph with a holding probability by simply doubling the length of the random walk.

To achieve good mixing, we would like to approximate w quite closely. For n peers, we would like the walk to approach w with proportional error of $\pm\delta$. If one reconsiders the random walk analysis from Section 6.1 to start at a node a, then $|V^{-0.5}x|$ is actually bounded by $\sqrt{1/\deg(a)}$. If one looks at the error in the single term for node b, the magnification by $|V^{0.5}|$ is just $\sqrt{\deg(b)}$. Since all peers joining have at least degree one (their bootstrap connection), the random walk can neglect the $\deg(a)$ term. Also, since we are interested in relative error, and a square-root is sub-linear, we can neglect the $\deg(b)$ term as well.

We know that $|\lambda_2| < 2\sqrt{d-1}/d < 2\sqrt{1/d} = 2\sqrt{1/16} = 1/2$. Therefore after m steps we would like,

$$2|\lambda_2|^m \leq 2(1/2)^m = \frac{\delta}{n}$$

Thus,

$$m := \frac{\log(\delta/2n)}{\log(1/2)} = \log_2 n + 1 - \log_2 \delta$$

Setting $\delta = 10^{-2}$, this simplifies to,

$$m \approx 7.64 + \log_2 n$$

Since we need to double this to compensate for the holding bias,

$$2m \approx 15.29 + 2\log_2 n$$

Thus, in BubbleStorm, a location walks $2m$ steps to pick a position in the Eulerian cycle. This suffices to approximate true uniform sampling to within 1% relative error.

7.2 Bootstrapping

In order to start a random walk, a location needs to know the identity of at least one participant in the network. If the peer already has neighbours, it just starts its random walks via that neighbour. However, if it has no neighbours it must begin the bootstrap process.

There are four ways a peer can enter a network:

- The peer can create a new network, containing only itself.
- The user can supply the address of a peer known to be in the network.
- The peer can contact peers listed as an entry-point in DNS.
- The peer can probe peers which were previously in the network (Section 7.2.2).

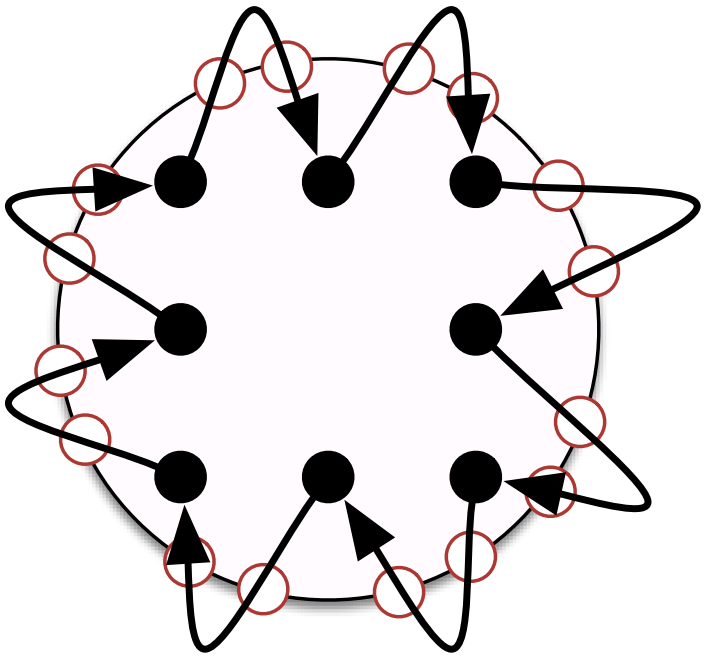


Figure 7.5.: The primordial BubbleStorm topology; 16 self-loop edges connect the 8 locations. Red circles indicate where the edges enter/exit the peer.

Creating a new network should only need to be done once by the application developer. The founder peer connects to itself, establishing a permutation of its locations; see Figure 7.5. Subsequently joining peers can partition the self-loops just as they would split any other edge in the system.

Once the network is up and running, it would partition the user base if a peer founds another network. As a practical matter, an application developer should probably not release an application which supports network bootstrapping.

7.2.1 Firewalls

In the modern Internet, firewalls often block unsolicited access to peers. While CUSP [82], our custom transport protocol, is able to punch a direct connection between two firewalled peers, this is only possible when those peers have an established side-channel. Thus, at the time a new peer wants to join the network, it cannot yet participate in the hole-punching protocol. Therefore, joining peers need to contact a peer which is not firewalled.

We explored several alternative side-channels in the scope of the BubbleStorm project. Unfortunately, none of them are wholly satisfactory. At the time of this writing, we have not finalized our choice; we still hope for a better option.

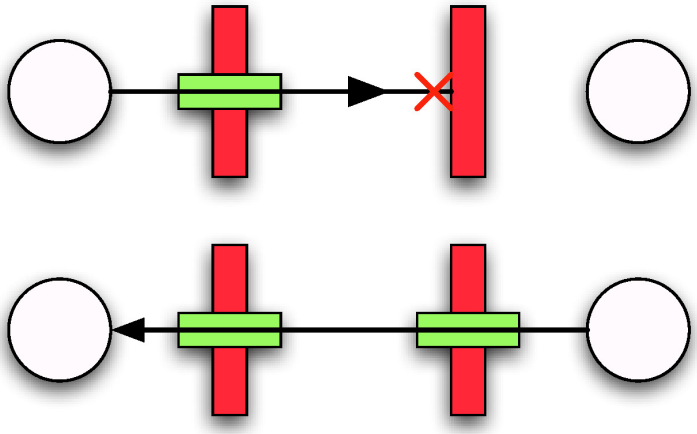


Figure 7.6.: The first peer to transmit is blocked by the other's firewall. The second peer to transmit makes it through, using the hole opened by the first.

Not every firewall can be penetrated by CUSP, but most are configured with a policy that allows the private network to access the external network, but not vice-versa. Another technology, called Network Address Translation (NAT [73]) causes the same effect as an artifact of its implementation. Unfortunately, due to the slow roll-out of IPv6 and paranoid administrators, most networks are now behind either a NAT or a "no external-to-internal access" firewall. This common setup offers some protection against hackers who would otherwise be able to connect to services/servers in the internal network. Clients inside the internal network can access servers in the Internet at large. Meanwhile, servers in the internal network are protected from access by clients in the Internet at large. Unfortunately, peer-to-peer applications simultaneously act as both client and server. These policies effectively prevent a peer from full participation.

Before explaining how we subvert this policy, it would be remiss not to discuss the ethics of this technology. It is my firm belief that the "no external access" policy is intended to protect potentially vulnerable internal services from attack. The policy allows all outgoing access, so the intention seems to be "let my users do what they please, but keep those bad guys off my fileserver." The firewall punching used in BubbleStorm only allows communication with peers running a BubbleStorm-powered application. Since an internal user had to start the BubbleStorm application, this seems to be tacit permission to allow that application to communicate with the BubbleStorm network at large. We never puncture a firewall which limits the access of internal users. If the policy doesn't include a "let users access what they please" rule, BubbleStorm applications will not be able to run, in accordance with the system administrator's policy. Thus, we consider penetrating firewalls and NATs a reasonable and ethical technology. If a system administrator truly intended to prevent peer-to-peer access, he need only forbid outbound access to the BubbleStorm port.

To connect two firewalled peers, both must initiate a connection to the other. After sending an outgoing message (over UDP) through a supported firewall, the firewall will accept UDP messages in response; see Figure 7.6. Since both peers sent a message, both their firewalls will now accept messages from each other. Thereafter CUSP can make data flow. The only difficulty is the (seemingly) easy task of convincing the two peers to send a message to each other.

In BubbleStorm, a query is answered by rendezvous peers which may never have communicated with the source peer before. Thus, they will need to puncture the firewall of the source peer to answer his query. Unfortunately, the source peer does not know the rendezvous peers, so he cannot send a message to open his firewall for the response. This is where we need a side-channel. To our knowledge there are three categories of IP side-channels: topology, destination-side, and source-forgery.

The first option is using the BubbleStorm topology as a side-channel. Here, the rendezvous peer routes a message back to the source peer over the same topology path that was used to contact him. Then the source peer and rendezvous peer know of each other and can directly connect. Aside from the obvious latency penalty, this approach lacks flexibility. A peer might want to connect to a peer it has learned about through some other mechanism. For example, it might issue a query to find online chat buddies. Then it would like to connect to those buddies, but it recovered the address from a rendezvous peer and not the buddy himself. While it might be possible to make this work, it is not a very clean approach and is likely to be quite failure prone.

The second option is using a helper peer connected to each firewalled destination peer. Supposing peer u is firewalled, he keeps a connection open to v who is not firewalled. When w wants to contact u, he asks v to tell u about him. Thereafter, u and w communicate directly. The problem here is that w needs to know about v. The address information for u is then (u, v) instead of just u. Thus, u advertises to everyone both u and v. Unfortunately, v might quit, causing u to find a new replacement helper v'. Now anyone who wants to contact u using the old address (u, v) is unable to. They need the address of the new helper v'. Again, it would be possible to make this work, but it would be failure prone.

The final option is to forge packets. Here every firewalled peer v contacts a non-existent, but well-known peer z. Then if u wants to contact v, he forges a message that appears to come from z. This message can enter v's firewall because v has already sent messages to z. Of all the options, this one has the least complexity. Peers only need to advertise their own addresses and direct connection establishment is possible. Unfortunately, forged packets have been abused by hackers in the past. Thus, most ISPs prevent users from forging the sender address on packets they send. There are two ways to improve this situation. First, unfiltered peers could provide a "packet forgery service" to the other peers. Unlike the previous intermediary approach, these helper peers are source-side and can thus be replaced without needing to propagate any new address information. Second, forged ICMP error packets are not yet as widely filtered [18] as simple UDP. While this approach is technically the best, we are unsure how wise it is to depend on the deficient filtering of ISPs.

Probably the best approach would be a packet "forgery" service using a unicast address. Connect several introducer systems around the world, all advertising the same IP block range via BGP/etc. Every new peer sends a message to the unicast address to open up a back-path. When a peer seeks to contact another peer, he sends his request to the unicast address, which arrives at the nearest introducer system. That introducer can then forward the connection request to the actual destination. Because the destination system already sent a message to the unicast address (though it was probably processed by a different introducer), the forwarded request from the introducer will penetrate the target firewall. The advantage of this scheme as compared to packet forgery is that the introducers actually use their own advertised addresses, the unicast address. Furthermore, because unicast routes to the closest introducer, very little latency penalty is imposed, less than half a round trip. Unfortunately, there is no installed base of introducer systems, so this approach can not be used today. It would be quite simple for a company like google to add this service to their unicast DNS servers, thus solving the NAT problem for everyone.

7.2.2 Host Cache

In order to rejoin the network, peers need a list of potential peers. The joining peer then probes several of these candidates to find one online. Our current implementation probes 6 at once with a 20s timeout before trying 6 more. Once a candidate bootstrap peer responds, that peer is used as the first step in the random walk to find locations to split (Section 7.1.2). The host cache provides the list of peers to start this process.

The general idea is that a peer records the addresses of other peers which are not behind a firewall. It prefers peers which have been online for a long time, assuming that this predicts that they will be online longer still. We give every peer in the host cache a score. Every time a peer is seen, its score is linearly increased (+0.1). Every time a peer is missing, it's score is multiplicatively decreased (90%). New addresses start with a score of 0.5. When joining, peers are contacted in decreasing score order.

We fill the host cache using the BubbleStorm API. Every hour peers execute a query for the addresses of bootstrap peers. Bootstrap peers advertise their availability using a managed bubble (Section 4).

There was an interesting attack used by Microsoft in their Microsoft Active Response for Security (MARS) Project. To combat peer-to-peer botnets, they essentially broke the same bootstrap algorithm used in BubbleStorm. First, they seized control of the DNS records used as well known points of entry. Then they poisoned the botnet's host cache [26]. This attack was effective because the host cache of the botnets was small. If defense against this sort of take-down is necessary, the host cache should be increased from the current 1000 to perhaps 1000000; addresses are cheap to store. One million addresses cost just 6MB.

7.3 Evaluation

Now that the components of the BubbleStorm topology have been detailed, we turn our attention to how well the implementation works.

To analyze the system, we built a simulation framework as part of the BubbleStorm project. The simulator design and implementation were largely the work of my colleague, Christof Leng, and are detailed in his thesis [49]. He explores the related work for network simulators and our design decisions.

To briefly summarize our reasoning, we decided to analyze BubbleStorm two ways. First, we designed the system such that we could use mathematical analysis to model and predict the expected behaviour. This work was largely mine, and the resulting theory is a major topic in this thesis. Second, we decided to simulate the network at the packet level. While our BubbleStorm implementation could be measured deployed on a real network, either in the Internet at large or in a small testbed, this is neither as flexible nor as repeatable as measuring a simulation. Over the years we developed several different prototypes of BubbleStorm and measured them with many different simulators, the simulator used here was designed and validated by my colleague in his thesis [49].

Our simulation approach narrows the system interface needed by BubbleStorm down to just a delayed event scheduler and UDP messaging. This required our implementation to build its own reliable transport protocol on top of UDP. This protocol, CUSP, is thus part of the simulation and captures the effects of timeouts and retransmissions. All packets are delayed and lost using the delay model outlined in [43], based on real measurement data of hosts positioned around the globe proportional to global Internet use. The simulator does consider queuing effects (and resulting packet drops), but only on the last mile uplink/downlink; there is no cross-traffic inside the network.

Nodes in our simulation stay connected for an average of one hour, distributed as an exponential random variable. The average uptime is 5%; there are actually 20000 nodes simulated, but only $\approx$ 1000 are active at a time. 10% of nodes which leave the network do so by crashing. To control the population during simulated events, there is an additional mask which can enable/disable nodes on demand. Finally, these nodes are producing either a search or publish request every 15 seconds, so the network is quite busy.

Population	DownSpeed	UpSpeed	Degree
2%	50 MBit	10 MBit	1280
3%	25 MBit	5 MBit	640
15%	16 MBit	1 MBit	128
20%	6 MBit	512KBit	64
20%	3 MBit	256KBit	32
20%	2 MBit	192KBit	24
20%	1 MBit	128KBit	16

Figure 7.7.: Bandwidth distribution in heterogeneous simulations

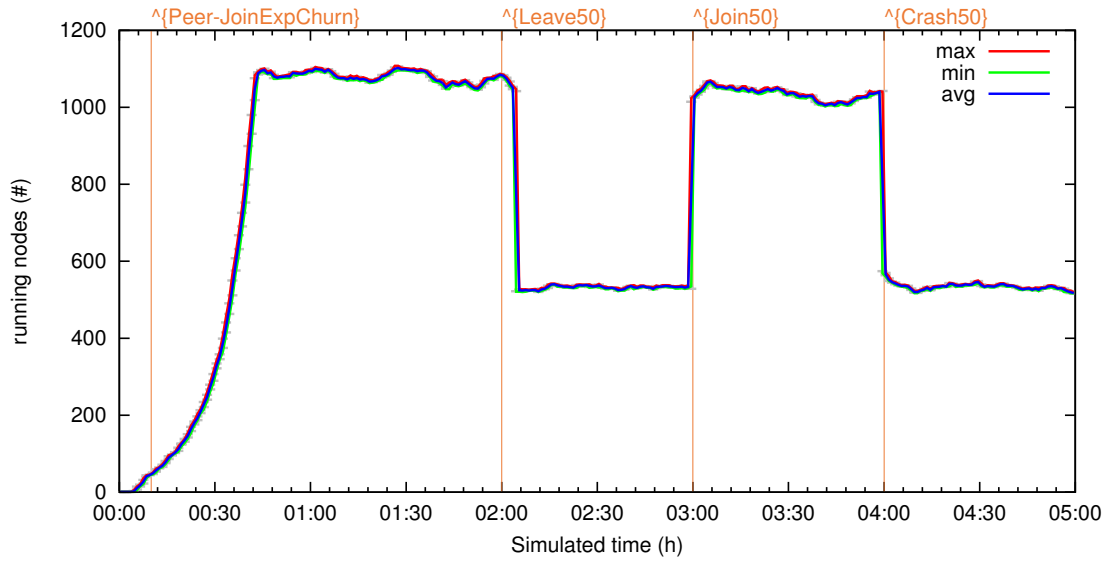


Figure 7.8.: Simulated node population resulting from exponential join, 50% simultaneous leave, 50% simultaneous join, and 50% simultaneous crash events.

The topology protocol's primary purpose is to support peers joining and leaving, keep the network connected, and tune node degrees. To examine its effectiveness, we run BubbleStorm on a network of 1000 peers with bandwidth as outlined in Figure 7.7. The network is initially grown with exponentially increasing arrival rate until the target population is reached. The network then runs under steady-state churn until the second hour, at which point half of the nodes decide to simultaneously start leaving the network, without crashing. After another hour, half of the nodes begin rejoining the network simultaneously. Finally, half of the nodes in the network crash simultaneously. The sequence of events is illustrated by Figure 7.8.

The join protocol requires the completion of a random walk. This causes the few seconds of delay in Figure 7.9. Initially, the network is small so the random walk is short. As the network grows in size, the join times grow logarithmically, which appears as linear on this graph due to the exponential rate of increase in network size.

Due to message loss from peer crash events, sometimes the random walk gets lost and must be retransmitted after 240 seconds. The topology protocol initially sets 8 locations to join and upgrades from client mode to peer mode (signalling join completion) once the degree reaches 4. In a perfect world, this means that only two walks need to succeed before the node becomes a full peer. If more than 6 walks go missing, however, then the peer must await the timeout before it completes joining. This can happen due to independent chance, but it can also happen when the used bootstrap peer crashes during the initial dissemination of the random walks. In any case, we can see this effect during the busiest part of exponential growth, during the peak join and leave events, and during the network doubling at hour 3. The crash event also causes a large number of join requests which were inflight to fail, leading to more timeouts.

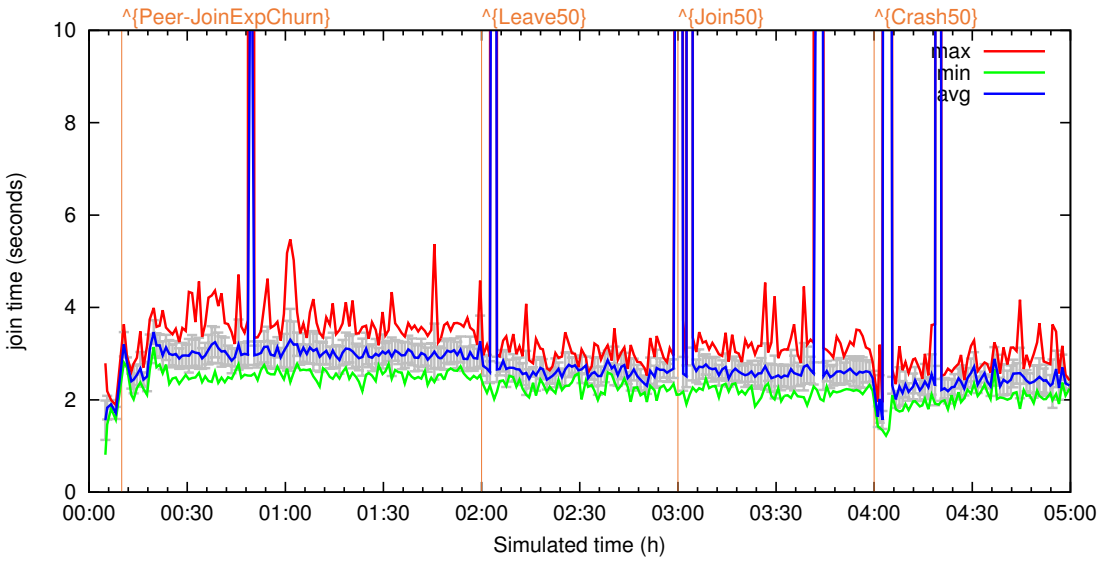


Figure 7.9.: Time required for peers to complete the join protocol and become full peers. The spikes correspond to (rare) unlucky nodes that must retry joining.

Keep in mind that nodes which have not completed the join protocol can still use the BubbleStorm network. They have client links to at least their initial bootstrap peer. These client links provide the peer with near immediate access to the rendezvous infrastructure. The join times simply measure the point at which the peer is able to contribute towards the processing of rendezvous workload.

To cleanly tear-down a location, a peer must ask its master to connect to its slave. After which, the leaving peer has two client links and the master and slave are directly connected. Of course, the master might be busy injecting a new peer into that location or be leaving itself, in which case the leaving peer has to wait for his new master before he can reissue the leave request. The master might also have crashed and thus never processes the request. Worse, the master might be waiting to insert/remove a different peer that crashed.

Unfortunately, in contrast to the join protocol, where a few successes are enough to declare completion, the leave protocol must tear down every location before completion. This greatly magnifies the chance that something goes wrong. As we see in Figure 7.10, the 20 second leave timeout plays a significant role. At this point, the peer ceases to wait for the master, and just resets the link. Sometimes leaving can take even longer as a peer restarts the 20 second timeout when its master is replaced. As the minimum shows, when all locations are able to exit cleanly, the process terminates quite quickly. In fact, one can nearly interpret the average leave time divided by 20 seconds as the probability that something goes wrong on at least one location while leaving. If an impatient user kills the slow-exiting BubbleStorm client, it is no worse than crash, i.e. well tolerated by the network.

These results may seem surprising when one considers how simple it really is to leave a location: one message to the master, asking to quit. The master (if not busy at this

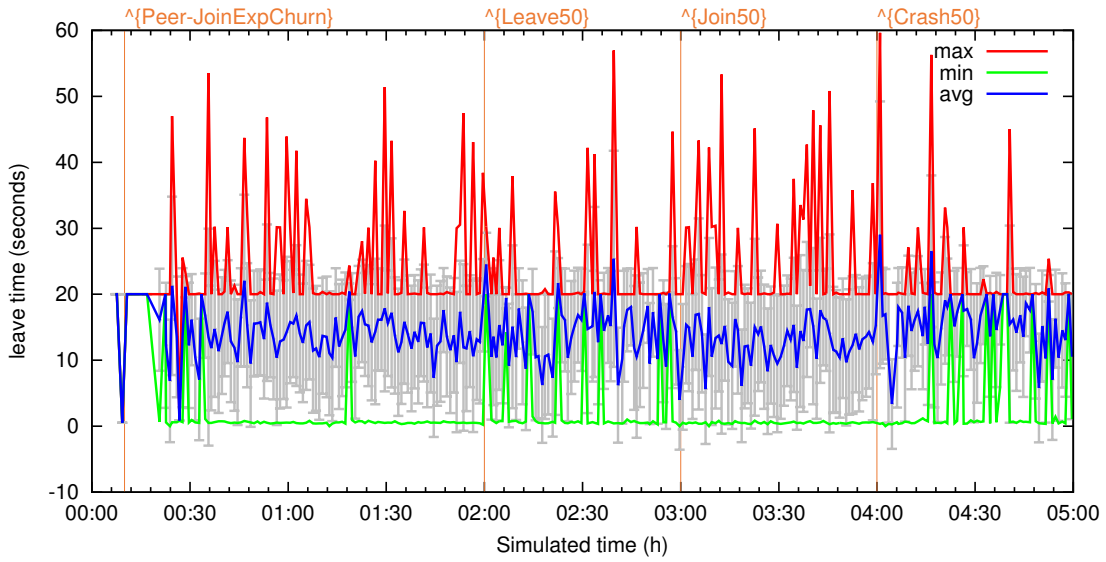


Figure 7.10.: Time required for peers to complete the leave protocol. The average delay corresponds to the chance of a single failure triggering the 20s timeout.

location) immediately confirms by closing the connection, making it half duplex, without waiting to successfully contact its new slave. In this simulation we still have publish/query rendezvous traffic (detailed in Section 9.4), and any inflight messages still need to be sent and acknowledged before the close is sent. Packet losses might lead to retransmissions and further delays. Hopefully, this plot demonstrates just how difficult it is to cleanly tear down a link, while maintaining a ring structure in an active system. Fortunately, BubbleStorm does not depend on ring integrity for correct operation, unlike some structured systems.

Unfortunately, the fun doesn't stop here. The underlying transport protocol (CUSP) must still deal with reliably delivering final acknowledgement information to all connection partners. Due to the two armies problem, this has to be achieved by a timeout, similar to the TIME_WAIT state in TCP. Although CUSP has some optimizations to minimize how often this is necessary, it still happens fairly often that the application cannot quit until the transport protocol timeout expires, which is mandated by the IETF to take at least 240s. Normally this would be handled behind the scenes by the operating system's TCP stack. However, since this is implemented as part of BubbleStorm itself, this can sometimes manifest in an additional 240s delay above and beyond the leave times shown in the topology plot. For these reasons, when asked to quit, BubbleStorm applications shut down their GUI and services promptly, and then switch to a background process with minimal memory footprint until all traffic is finalized. If it were possible to hand-over the task of final acknowledgement to the operating system's kernel (as TCP enjoys), this background process delay could be avoided.

To explore how quickly the topology recovers desired degree, we turn our attention to the simulation in Figure 7.11 where the peers all have 1MBit instead of the heterogeneous distribution ala Figure 7.7. This way, all the peers have the same target degree

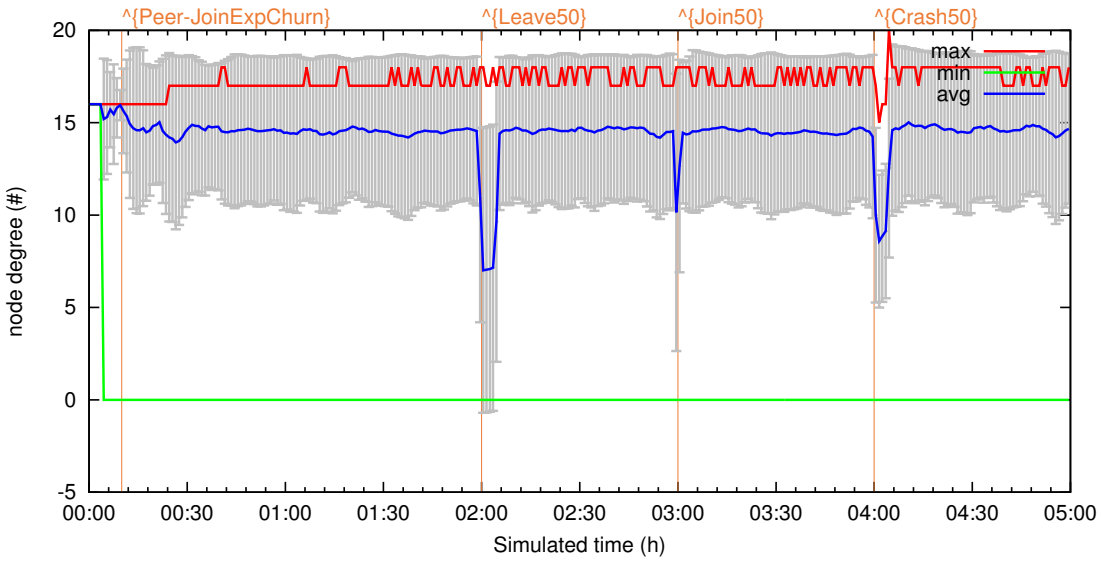


Figure 7.11.: Node degree in a homogeneous network (including leaving and joining nodes). Maintenance quickly restores the system to the desired degree.

making the plot a bit easier to interpret (the heterogeneous plot is similar, but with larger variance due to the different desired degrees).

We can see that the minimum degree is nearly always 0. This is due to churn; there is almost always at least one peer trying to join the network at any given time. The maximum degree can briefly exceed the 17 neighbour cap, when a half-broken location acquires an unexpected new neighbour due to a join. Thereafter, the degree tuner restores the degree fairly quickly.

When half of the peers leave the network, they steadily reduce their degrees as their masters honour the leave requests. Figure 7.11 is an average over all running peers, including those in the process of leaving. Thus, since it takes time to leave, the plot shows a dip in average degree. Nevertheless, the topology protocol is working correctly and all the non-leaving peers retain the correct number of neighbours during this procedure due to the topology protocol. The same situation is visible on the mass join; nodes can become peers as soon as they reach degree ≥ 4, temporarily affecting the distribution. The crash event causes the degree distribution to drop since half of the edges in the system are broken. This is *not* a measurement artefact, as crashed peers are not included in the average. The reduction in peer degree on crash is due to actual broken edges. Over time, the non-crashed peers detect the crash and acquire new neighbours via the random walk protocol.

Finally, we consider the traffic requirements of the topology protocol. This breakdown considers only the message payload traffic; overhead due to CUSP, UDP, and IP headers, while simulated, is not measured. Due to the nature of our transport protocol, multiple messages are packed into a single packet and it is thus not possible to meaningfully assign these overheads to messages. We split the traffic into four categories: bubblecast,

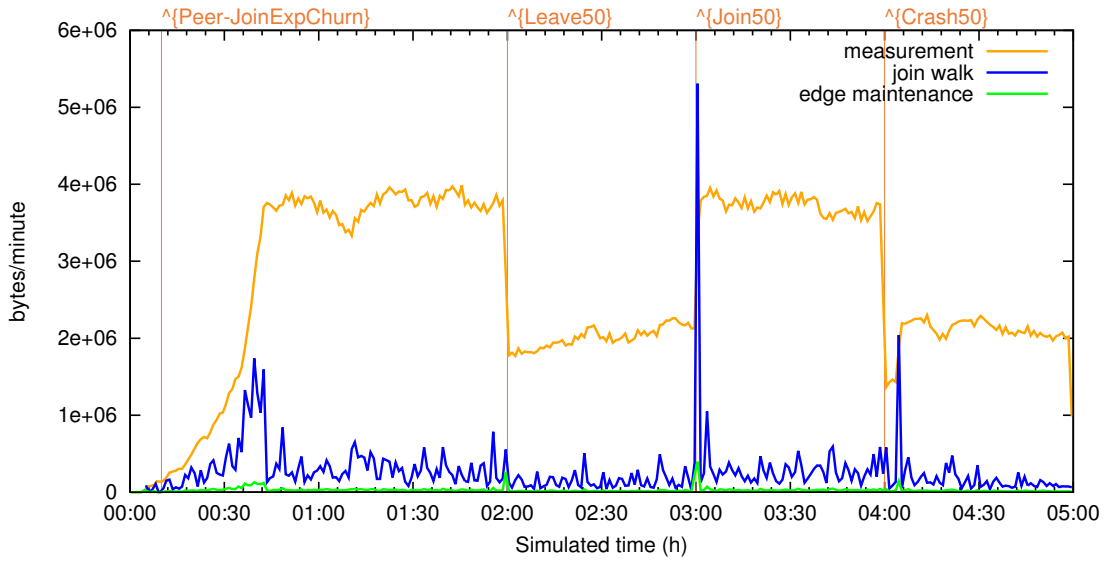


Figure 7.12.: Traffic breakdown for a heterogeneous network. Measurement traffic dominates except during major topology changing events.

measurement, join random walk, and edge maintenance. As we will see in Figure 9.9, bubblecast traffic dominates all others and so we remove it from consideration here.

Comparing Figures 7.12 and 7.13, we see that the heterogeneous network has much larger fluctuations. This is because the high bandwidth peers have a large effect on the network when they join or leave. Also, the traffic is about 5 times higher for the heterogeneous network due to the larger number of edges in the network. The shape of the graphs in response to the simulated events is the same. There aren't any major surprises; the topology traffic is higher when the churn rate is higher. After the crash event, the dead link timeouts all fire more-or-less at once, causing many peers to execute walks to refill their missing degree.

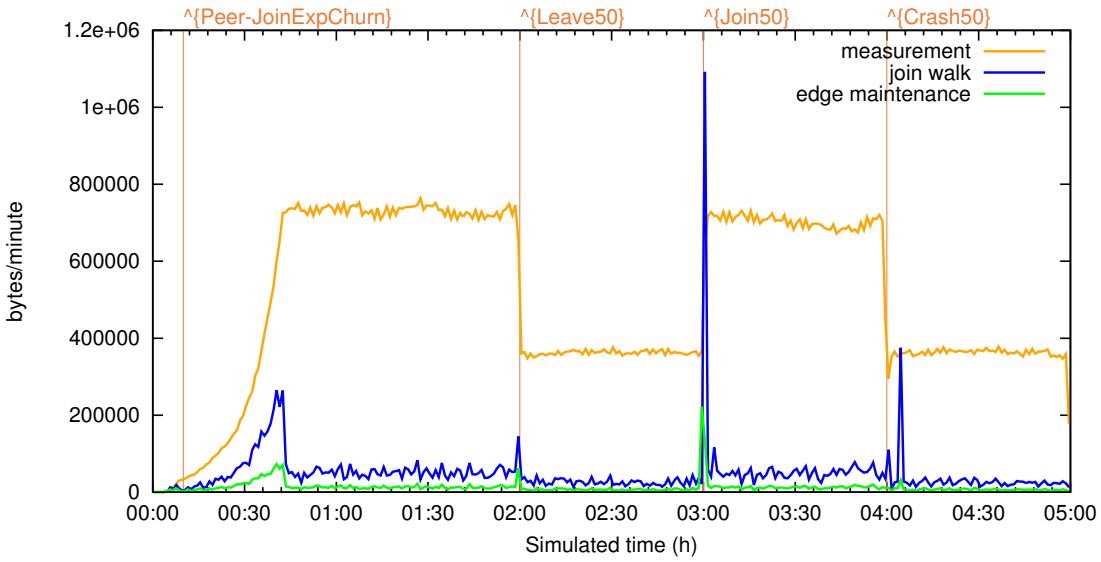


Figure 7.13.: Traffic breakdown for a homogeneous network. Less traffic than a heterogeneous network, but with a similar proportional breakdown.

8 Measurement Protocol

To ensure the correct rendezvous probability, the bubble balancer (Section 5) needs to ensure Equation 5.1 holds, while minimizing $\sum_{t \in T} S_t x_t$ (Equation 5.2). This requires the terms S_t for every bubble type $t \in T$ plus w_m and $\sum_{u \in U} w_u^2$. These last terms can be rewritten as,

$$\begin{aligned} w_m &= \max_{u \in U} w_u = \frac{\max_{u \in U} C_u}{\sum_{u \in U} C_u} = \frac{\max_{u \in U} \deg(u)}{\sum_{u \in U} \deg(u)} = \frac{D_\infty}{D_1} \\ \sum_{u \in U} w_u^2 &= \frac{\sum_{u \in U} C_u^2}{\left(\sum_{u \in U} C_u\right)^2} = \frac{\sum_{u \in U} \deg(u)^2}{\left(\sum_{u \in U} \deg(u)\right)^2} = \frac{D_2}{D_1^2} \end{aligned}$$

using the following definition of D_i,

$$\begin{aligned} D_i &:= \sum_{u \in U} \deg(u)^i \\ D_\infty &:= \max_{u \in U} \deg(u) \end{aligned}$$

Similarly, S_t can be found by summing the bubble-type-t traffic generated by each peer. Thus, all the terms needed can be expressed in terms of either the sum or maximum of values local to each peer.

The measurement protocol is responsible for gathering this global knowledge. In the measurement protocol, every peer contributes a real-valued number. After the protocol completes, every peer knows the sum of all the contributed numbers. The goal is to approximate the true value of the sum as closely as possible, despite the constantly changing membership of the network.

For BubbleStorm, due to the churn of peers joining and leaving the system, the actual values of D_i are constantly changing. However, the overall population distribution of a peer-to-peer network changes slower than the raw turnover rate of the peers. Measurements of deployed peer-to-peer systems [37, 71, 76] show the population changes mostly with a 24-hour rhythm. Therefore, for practical use we just need a slowly moving approximation; an updated value in the range of every 5 minutes is good enough.

There are several approaches one can take to the measurement problem. Probably the simplest to understand is an aggregation tree. Each leaf forwards its contribution to its parent. Once an intermediate node has the contribution of all its children, it too forwards the result to its parent. Eventually, the root has received all the values, and then redistributes this information down the tree. It's fairly obvious that this approach is asymptotically optimal in both bandwidth $O(n)$ and latency $O(\log n)$. However, because a real network is constantly in flux, one cannot guarantee that any tree stays intact long enough for the values to be accumulated. Nevertheless, some structured systems do try this approach, like Cone [11].

Gossip protocols [39] take another approach. Here, every peer exchanges information with its neighbours. In contrast to the tree aggregation approach, every peer speaks in every round. An easy to understand scenario would be finding the maximum value. Every peer tells all of its neighbours the largest value it has yet seen. This process is repeated every round until the result stops changing. In this way, the true largest value is flooded from its point of origin. Like the tree aggregation protocol, the maximum reaches all peers after logarithmically many rounds. However, it uses $O(n \log_{\lambda_2} n)$ traffic as opposed to $O(n)$. In trade for this traffic, the procedure is much more robust; the algorithm will always succeed as long as the graph stays connected.

Computing sums with gossip is slightly more complicated. There are two approaches, both interesting for BubbleStorm. The first approach builds on top of the minimum finding algorithm. Each peer samples an exponential random variable, with a parameter (the inverse of the mean) equal to its value to contribute. A useful feature of exponential random variables is that the minimum of the several variables is itself exponentially distributed, with a parameter equal to the sum of parameters. Thus, if every peer contributes one sample from an exponential random variable and the global minimum is found, the result is a sample from a new exponential random variable with parameter equal to the sum. Unfortunately, just one sample does not give a very good estimate on the parameter of the resulting distribution. To pin down the actual value, the procedure must be repeated. A simple variation has each peer sample the distribution multiple times in parallel [60], thus one execution of the gossip protocol obtains several samples.

While we did not use this randomized algorithm for the measurement protocol, we have considered using it for some other statistics. For example, suppose one wanted to calculate the total number of unique files in a network. Just adding up the files at each host would give the total files, but not the total unique files, because the same file gets counted once for each peer which stores it. Instead, one could use the hash of a file as a random seed for an exponential random variable. Each peer locally finds the minimum of one sample taken per file (using that file's hash). Once the global minimum is found, this will be a sample from a distribution with parameter equal to the number of distinct seeds, or hashes, or files, used.

The other approach to finding sums using gossip is to rely on mixing. Recall from Section 6.1 that a random walk eventually converges to a stable distribution no matter where it starts. Identically, if each peer were to divide its value between its neighbours each round, the steady state would spread the value across the network proportional to degree. Unfortunately, due to the impracticality of perfectly synchronized rounds and a perfectly static topology, this approach doesn't work. The trick to using this in peer-to-peer comes from Kempe et al. [44]. Instead of mixing a single value, he mixes two. One value is the value to measure, call it water, while the other value serves as a measuring stick, call it salt. If you mix both values and stir, the salt will eventually be distributed uniformly throughout the water. As long as you know the total quantity of salt, you can extrapolate the total water in the system from the density of the salt in the water you currently have. This is the approach BubbleStorm takes.

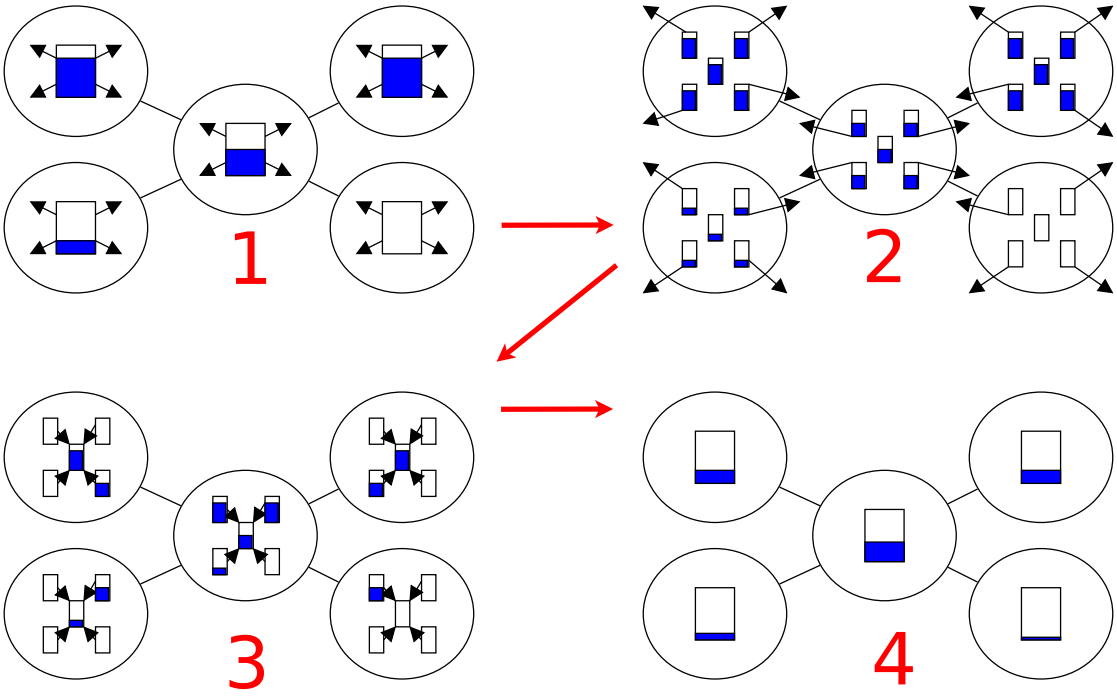


Figure 8.1.: Kempe's water mixing algorithm shown for 5 nodes embedded in a larger 4-regular graph. Divide a peer's water into smaller jars. Exchange those jars with neighbours. Then, recombine the jars ready for the next round.

8.1 Approach

In Kempe's two variable mixing algorithm [44], each peer has a bucket of water[1]. Initially, each peer fills its bucket with the value it contributes to the sum. One specially chosen peer adds 1kg of salt to its bucket. In each step of the algorithm, the peers divide their water equally between cups, one for each neighbour and one for themselves. They then send their cups to their neighbours and receive their neighbours cups in return; see Figure 8.1. All the received cups are dumped back into the bucket and stirred. Kempe showed that eventually this process will fully mix the salt within the water. When the algorithm terminates, each peer will have a different local amount of water and salt, but the ratio of salt to water is the same everywhere. Thus, peers can simply solve the following equation for the total water in the system (the sum of all initial water),

$$\frac{\text{my salt}}{\text{my water}} = \frac{\text{total salt}}{\text{total water}} = \frac{\text{1kg}}{\text{total water}}$$

We build on Kempe's algorithm in several ways. Firstly, we remove the unrealistic assumption of synchronized gossip rounds. For the bubble balancer, salt water is shared with neighbours once every 90 seconds. Thus, if the peer has 10 neighbours, it sends a

[1] The salt water analogy is ours. The original paper is straight-up math.

gossip message to one of its neighbours every 9 seconds. The initial order neighbours are contacted is arbitrary, but we re-contact them in a round-robin fashion. The clocks of peers need only run at roughly the same speed. No attempt is made to synchronize the message exchanges; every peer autonomously sends a gossip message according to its own schedule.

The measurement protocol runs continuously. Once it has calculated a sum, it immediately begins calculating the sum again. Any peer present when the algorithm begins contributes its value to the calculation. Every peer online during execution participates in the mixing. To synchronize when the next calculation begins, BubbleStorm gossip messages include a sequence number. A peer reinitializes its contribution with the next sequence number when either: a) it received a message with a larger sequence number, or b) the result has been stable to 64ϵ (ϵ is precision of a floating point number) for one exchange with every neighbour, plus another 16 steps for safety.

To implement Kempe's special, salt donating, peer, every peer generates a 64-bit pseudorandom number from a seed based on its IP+port and the sequence number. Due to the hash function chosen, there can be no collisions and there is a unique maximum somewhere in the network. This random number is included in all gossip messages. When the measurement begins, every participating peer contributes salt as if it were the designated special peer. However, peers only accept salt that corresponds to the largest random number seen. Thus, eventually, only the salt from the peer with the largest number remains.

Finally, we need to decide how much water and salt to share with our neighbours. We could stick with Kempe's algorithm where u sends $1/\deg(u)$ of its salt water. However, we will see that this is quite slow. This approach only makes sense if there is a synchronized round switch where all peers exchange a message with all neighbours simultaneously. To find the best amount to share, we measure how much the global error is reduced for a single mixing. We will take the greedy approach of maximizing each individual mixing.

Consider two quantities u, v of water and a, b of salt to be mixed. Suppose the true average (total water in the system) is x. We can ask how much these two peers contribute to the variance from the true mean, weighted by their salt content,

$$a(u/a-x)^2+b(v/b-x)^2=\frac{(a+b)(bu^2+av^2)}{(a+b)ab}-2ux-2vx+x^2(a+b) \qquad (8.1)$$

After mixing, the new estimate will be $(u+v)/(a+b)$. Thus the contribution to the global error is,

$$(a+b)((u+v)/(a+b)-x)^2=\frac{ab(u+v)^2}{(a+b)ab}-2ux-2vx+x^2(a+b) \qquad (8.2)$$

We can now subtract 8.2 from 8.1 to find the improvement,

$$\frac{(a+b)(bu^2+av^2)-ab(u+v)^2}{(a+b)ab}=(u/a-v/b)\frac{ab}{a+b} \qquad (8.3)$$

Inspecting 8.3, we can see that the improvement depends on two things. First, $u/a - v/b$ is the difference in the salt densities of the waters; so, the improvement in mixing is proportional to the current mixing. Thus, we can expect that the overall system will converge exponentially. The second term is more interesting. It relates the improvement to the relative amount of salt the two peers contribute.

Suppose that you have a fixed amount of salt S to split between the two nodes. Then, $ab/(a+b)$ is maximized when $a = b = S/2$. So, we would like to mix equal amounts of salt together, to maximize mixing. Indeed, as we will see for a homogeneous network, this is borne out in practice.

Consider instead the situation where you have different degree peers. The higher degree peers mix more frequently, so it seems intuitively obvious you would like more salt mixed on them. To quantify this, imagine one source of salt a mixed x times with x other sources of salt b. You want to split your S between these $x+1$ sources, $S = a+xb$. Maximizing $ab/(a+b) = (S-xb)b/(S-xb+b)$ is a job for calculus. Take the derivative with respect to b and set it equal zero to find the maximum,

$$\frac{(S-2xb)(S-xb+b)-(1-x)(S-xb)b}{(S-xb+b)^2} = 0$$

It is easy to verify that $b = S/(x+\sqrt{x})$ is the solution, so $a = S\sqrt{x}/(x+\sqrt{x})$.

While this solution looks complicated, if one ignores the denominator, the result actually just says to weight by the square-root of the mixing rate. This matches our intuitive expectation that high degree peers should hold more salt to leverage the fact that they mix more often, although it is admittedly difficult to justify the square-root. Also notice that this matches the homogeneous result; if the peers all mix with the same rate, then the salt should be shared evenly.

To achieve this ideal mixing, we need a way to bias how much salt peers hold in the steady-state. This global steady state can be achieved by carefully choosing our local mixing rule. Instead of u sending its neighbours $1/\deg(u)$ water, we choose that u sends v water according to,

$$\frac{\sqrt{\deg(v)}}{\sqrt{\deg(v)}+\sqrt{\deg(u)}} \tag{8.4}$$

To verify that this has the desired steady state, consider a simplified scenario where the water exchanges between two peers are synchronized. If peers u and v have $S\sqrt{\deg(u)}$ and $S\sqrt{\deg(v)}$ salt respectively, they will both send each other $S\sqrt{\deg(u)\deg(v)}/(\sqrt{\deg(u)}+\sqrt{\deg(v)})$ salt. Thus their salt content is unchanged, and $S\sqrt{\deg(u)}$ for all u is the steady state.

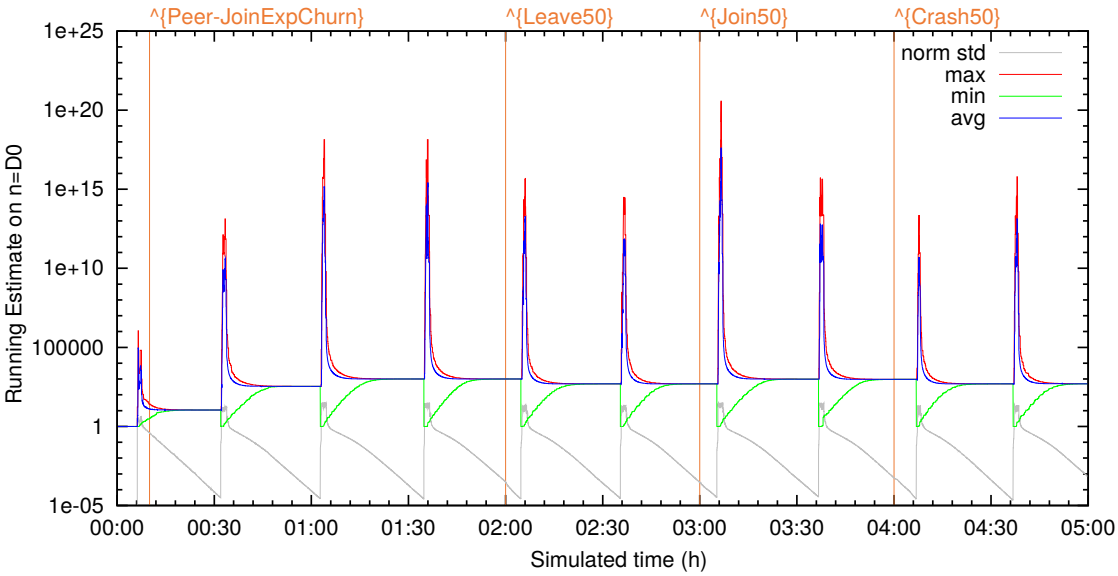


Figure 8.2.: Kempe's mixing algorithm, run to compute $n = D_0$ on a homogeneous network. The minimum and maximum quickly converge to the correct value, while the normalized standard deviation falls exponentially.

8.2 Evaluation

In the preceding section, we made two simplifying assumptions. We assumed that optimizing each mixing step individually was a good idea. We also modeled our problem assuming a synchronous water exchange between two peers. Thus, one might argue that our mixing algorithm is suspect. Fortunately, our experimental results will show that our mixing equation is good.

Consider again the simulation used in Section 7.3. We will examine first Kempe's algorithm in a homogeneous network; Figure 8.2. This plot shows BubbleStorm's running estimate on $D_0 = n$, the size of the network on a logarithmic scale.

The first thing to notice is that the protocol begins with very different values for the minimum and maximum estimates in the network. These quickly converge towards each other. The normalized standard deviation of all peers from the true value is also shown. This sawtooth curve appears to decrease linearly due to the logarithmic scale; thus, it is indeed exponentially decreasing as expected in Section 8.1. Because it is normalized, we can see that when the round switches, the relative error is only 1e-5. Floating point numbers have $\epsilon \approx 1.2e-07$, so the round is switching a bit early. This is because the round switch rule was tuned for our own much faster converging algorithm.

Also worth noting is that the initial convergence is especially rapid. This corresponds to the propagation of the salt with the largest pseudo-random value. Until this salt has fully propagated throughout the network, the total salt in the system is actually greater than one, which can initially give a wildly overestimated result.

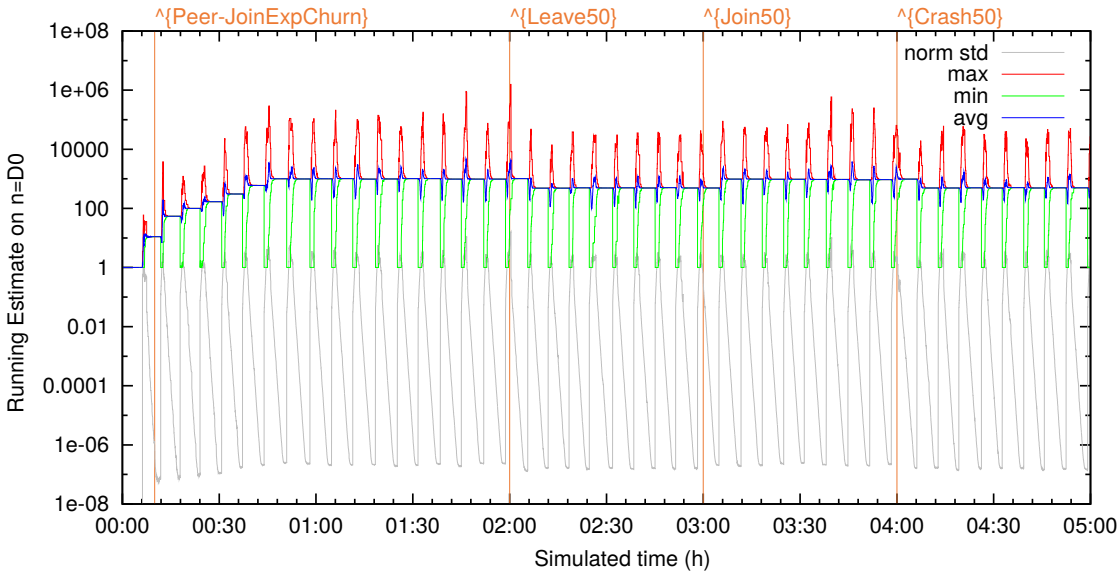


Figure 8.3.: The 1/2 mixing algorithm on a homogeneous network converges much more rapidly. The slope of the standard deviation is significantly higher.

The changing size of the network can be seen in the different values to which D_0 converges. However, on this logarithmic scale, even a doubling is quite a subtle change. The initial exponential growth is much easier to see. It is important to note that this network is undergoing constant churn with node arrivals and departures. Even the large-scale events where half of the network joins or leaves do not significantly impact the convergence of the algorithm.

The next plot in Figure 8.3 shows the result of changing the mixing rule. Instead of $1/\deg(u)$, this graph shows the result of sharing $1/2$. Keep in mind that $1/2$ is the degenerate homogeneous form of equation 8.4. The simulations in Figure 8.2 and 8.3 are both run with the same number of gossip messages, water shared once with every peer every 90 seconds. Nevertheless, the convergence is significantly more rapid. In a one hour interval, Kempe's rule terminates two times, whereas this rule terminates ten times. Furthermore, the final accuracy in our algorithm is near floating point precision.

Consider that in an hour, there are 40 90-second intervals. Since our algorithm completes 10 times, this means the algorithm only exchanged messages with neighbours 4 times before completion. If we had stuck to synchronous message rounds as in the original algorithm, this would only have been 4 rounds, nowhere near enough to complete mixing. Thus, our decision to send one message to one neighbour at a time has greatly improved the efficiency of the measurement protocol, not only its practicality of implementation.

Next, we turn our attention to a heterogeneous network in Figure 8.4. This mixing was performed with the $1/2$ rule instead of equation 8.4. As there are now peers with higher degree, there are more gossip messages being exchanged. This accounts for the 17 completed measurements per hour, instead of merely 10. Comparing Figures 7.12 and 7.13, the traffic appears roughly 5 times higher. Unfortunately, convergence is now

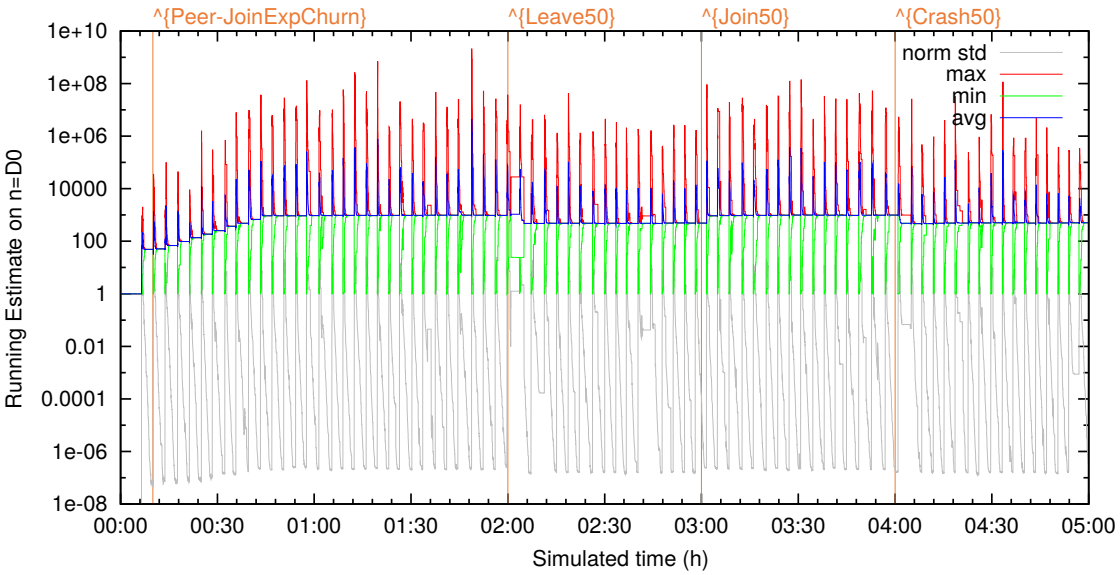


Figure 8.4.: The 1/2 mixing algorithm, when run on a heterogeneous network, has more edges in the graph, resulting in more frequent message exchanges.

so rapid that the topology protocol can confuse the results. It takes time for a peer to join/leave the network and it may be only partly connected during the procedure. In particular, leaving peers do not receive further gossip messages and so their estimate does not converge. Furthermore, peers can have a lot more edges and thus leaving can take even longer. These two effects combine to create rare situations where the convergence appears to stop. However, in reality, the active participants in the protocol did converge, else the round switch would not have occurred.

Finally, the result of using the heterogeneous square-root mixing rule, Equation 8.4, is shown in Figure 8.5. Now there are 22 completed measurements per hour. For reference, this means that it takes less than two exchanges between neighbours to achieve floating point accuracy. To put that in perspective, that's equivalent to just two broadcasts flooded through the network. While it's still true that this algorithm costs $O(n \log_{\lambda_2} n)$ traffic, it's hard to be particularly concerned about the extra log factor compared to a structured tree aggregation approach. In exchange, the algorithm is very simple to implement and continues to work in the face of massive network disruptions.

As a footnote, we have tried other mixing algorithms not shown here. For a homogeneous network, values close to 1/2 all seem to perform equally well up to measurement error. When one deviates too far in either direction, performance decreases. On heterogeneous networks, we used to scale salt proportionally to degree. This performs similarly to the mathematically justified square-root approach. However, its convergence is more heavily affected by the crashes of large peers.

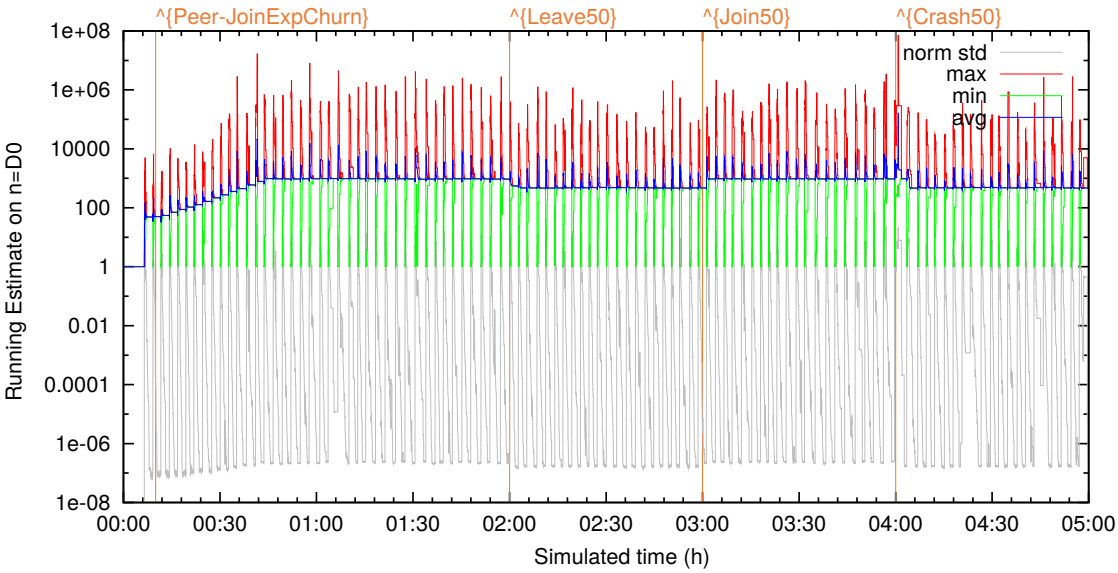


Figure 8.5.: Our mixing algorithm, based on Equation 8.4, run over a heterogeneous network converges significantly faster than all competitors.

9 Bubblecast

BubbleStorm spends a lot of effort to find the right size of a bubble. As we've shown in Theorem 8, it achieves minimal traffic consumption in large networks when these sizes are used. The bubblecast protocol is responsible for ensuring the correct number of replicas are placed in the network. The design goals for bubblecast are:

- Precise replica placement rate
- Balanced network utilization
- Low latency search
- Simple to implement

The simplest unstructured search systems flood using a hop count [42]. Replica placement messages include a counter. When the count is zero, a peer does not forward the message. If the hop count is positive, it is decremented and forwarded to all neighbours.

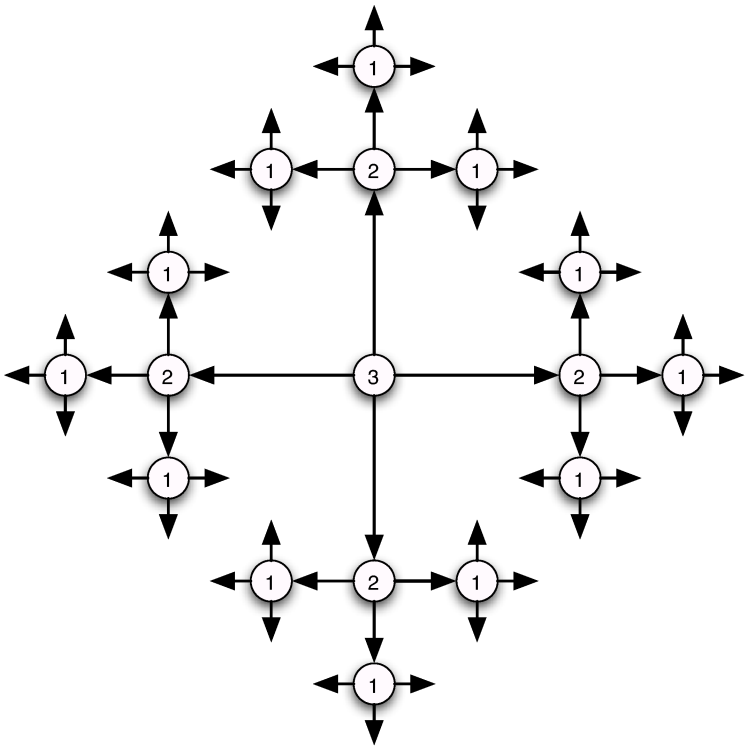


Figure 9.1.: Propagation of replicas using flooding on a degree 4 graph

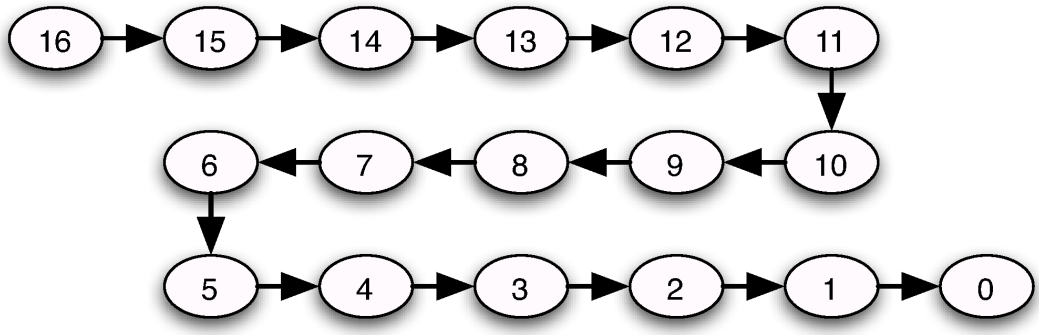


Figure 9.2.: Propagation of replicas using a random walk

Unfortunately, this is not very precise. It is only possible to control the logarithm of the replicas placed. Furthermore, if the network includes heterogeneous degree peers like BubbleStorm, then the number of replicas placed by the same initial hop count can differ depending on where the message originated.

An alternative is to use random walks [28,35]. Messages again include a counter, but this time messages are only forwarded to one other neighbour; see Figure 9.2. This way the message is forwarded exactly as many times as the initial counter value. This level of precision is what BubbleStorm needs.

Another benefit of the random walk approach is that it balances traffic. As we have already seen in Section 6.1, random walks approach a steady-state probability distribution. The first eigenvector, the steady state, has weight proportional to degree. If one peer has twice the bandwidth, it has twice the degree and thus twice the steady-state weight. Thus, a random walk uniformly utilizes the fractional capacity of peers.

Of course, if the traffic all originated on one peer, that peer would have load well in excess of the steady state. Still, after a few hops, the low second eigenvalue (i.e. fast mixing) of the BubbleStorm graph will have ensured that the random walks are well distributed throughout the network. Therefore, the load of the network at large will be balanced even though local to the traffic source there was some extra load. Furthermore, in a real system, traffic over time originates from many peers, not just one. Thus, the initial probability distribution is already very close to the steady state. Imagine a homogeneous network where peers all inject traffic at the same rate. Then the initial probability distribution is already the steady state. In a more heterogeneous system, the situation is probably more skewed, but the low second eigenvalue closes the gap to ideal load balance quickly. We will measure the simulated traffic balance of BubbleStorm in Section 9.4.

Traffic balance in flooding is very poor. The core problem is a double-dependency on degree. High degree peers have more neighbours and thus receive more traffic. However, they also replicate that traffic to all their neighbours, an $O(n^2)$ outgoing traffic rate. Compare this to a random walk where they receive similar in-bound traffic and forward to only one peer, $O(n)$. Truthfully, things aren't quite this simple. With flooding,

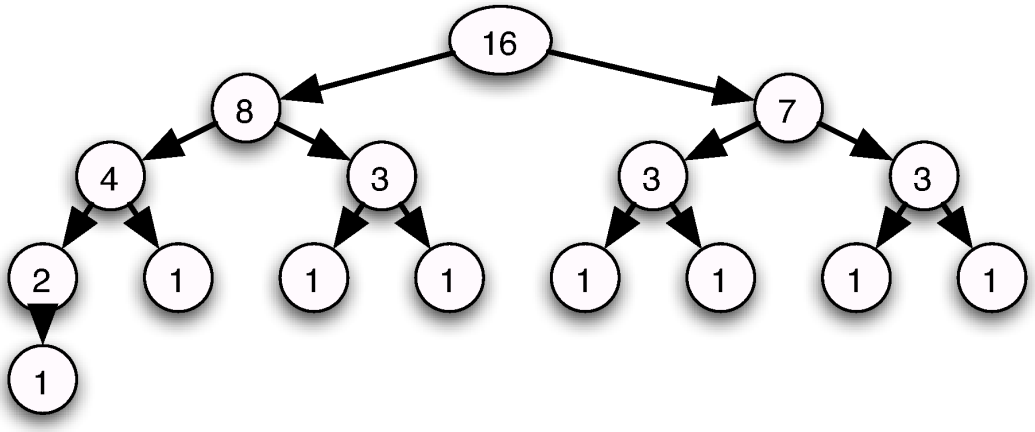


Figure 9.3.: Propagation of replicas using bubblecast

your neighbours won't all send you the same amount of traffic. High degree neighbours send you more. Thus a particular peer's load depends not only on its degree squared, but also its neighbours degrees, and their neighbours degrees, etc. Random walks are simple; load proportional to degree, period.

While flooding has poor traffic balance and imprecise replication count, it has two key advantages: latency and reliability. To reach 132 peers, it takes a flooding scheme only 2 hops in a homogeneous BubbleStorm network. In contrast, the random walk needs 132 hops. This huge difference in routing depth makes random walk query times much too slow. We will examine the impact of these choices on delays in Section 9.4.

Not only does random walk replica placement take far longer, but it is also less reliable. If a peer crashes after receiving a message, but before forwarding it, not only is that replica lost, but also any replicas which should have resulted from forwarding. Suppose that a flooding and random walk algorithm seek to place the same number of replicas. The chance of any message being lost is the same. However, the impact of that loss is much worse in a random walk where the average counter value is $n/2$ compared to $\log(n)$ for the flooding tree.

Bubblecast tries to find a balance between the two extremes of flooding and random walks. It too includes a counter, which is decremented upon message receipt. Like a random walk, the initial value of the counter precisely specifies the number of replicas. Unlike either flooding or walking, bubblecast forwards to two neighbours, regardless of degree. To maintain precise replication, the remaining value of the counter is divided by two. Figure 9.3 illustrates this procedure.

Certainly bubblecast is simple to implement and achieves precise replica placement. As for latency, the tree-like branching ensures that path lengths are short compared to random walks. They are not as short as flooding, where each level of the tree branches more widely, but in exchange we gain load balance. When the system is under load, this improved balance will reduce queuing in the network and thus improve latency. Still,

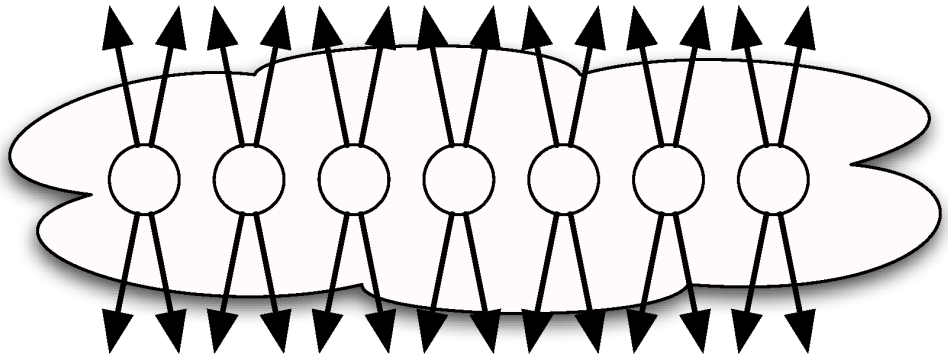


Figure 9.4.: Randomly selected peers have all their edges open to searches

on idle networks we indeed pay a latency cost for this 2-way branching. The reason 2-way branching yields load balance is intuitively simple: the load out only depends on the degree-in, $O(n)$ like a random walk.

To analyze the balance phenomenon a bit more carefully, we turn again to the mixing from Section 6.1. Suppose we wanted to measure the expected number of replicas a peer sees. We could use $2GV^{-1}$ to represent each step of bubblecast. The same eigenvector analysis will conclude that the resulting steady-state has load balanced according to degree, just with a factor 2^i for depth i routing. While $i = \log(x)$ may not be deep enough to fully mix a single traffic source, we can again argue that since the load is injected from many places in the network, it is already close to the steady state and thus few iterations are needed to reach the steady state. In any case, Section 9.4 will demonstrate this empirically.

9.1 Topological Dependency

An unfortunate side-effect of bubblecast (or any topology-based forwarding scheme) is that replica placement is not truly random. While we chose a random graph to approximate random placement, problems appear when the same graph is used for both data and query replication.

Figures 9.4 and 9.5 show one of the problems. Define interior edges as those connecting two peers inside the bubblecast tree and exterior edges as incident only on one peer inside the tree. True random data placement results in more exterior edges. When a query is executed on the same topology, bubblecast follows edges at random. Because there are fewer exterior edges on a data bubble than there should be, it is less likely that a query will select an edge leading to a data replica. This effect is what I've termed topological dependency, and it will affect any scheme which forwards both query and data replicas from one neighbour to another over a single topology. Flooding, random walks, bubblecast—all are affected. Figure 9.6 shows that the particular tree does not matter; both have 16 external edges and 6 internal.

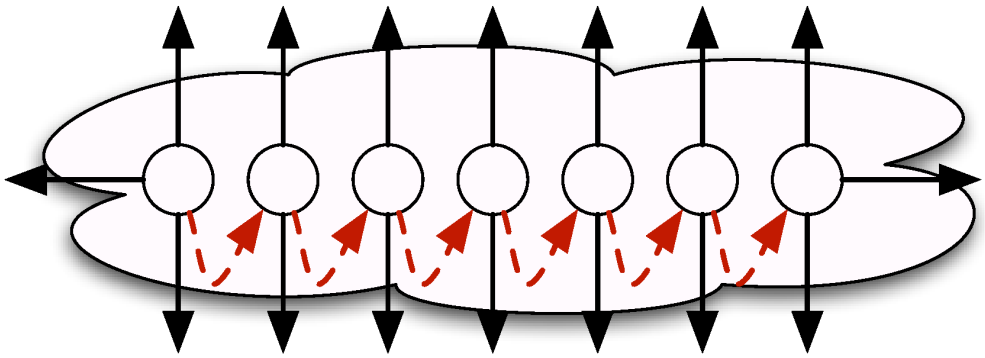


Figure 9.5.: Selection by a random walk causes fewer edges facing searches

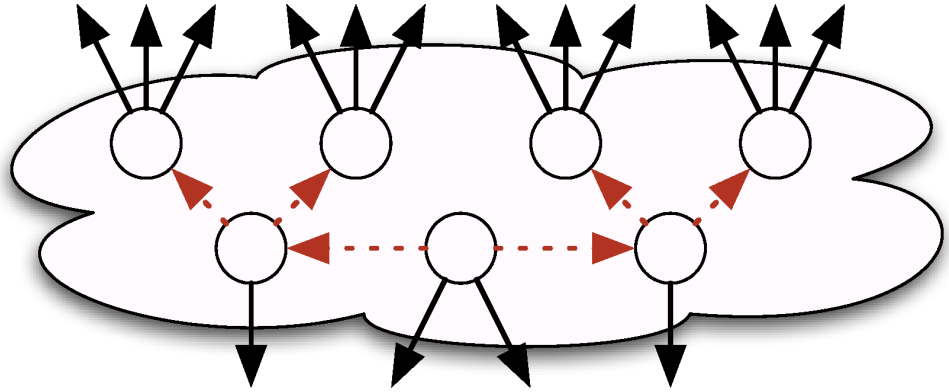


Figure 9.6.: The situation is not improved by bubblecast

There are a few ways we can avoid this effect. We could use two distinct networks, one for data and one for query bubblecast replication. Then, the exterior data edges in the query network are unrelated to initial data bubblecast and there is no excess of interior edges. Of course, this would be problematic in the real BubbleStorm system where there are many bubble types, any pair of which can be subjected to an intersection constraint. Alternately, we could use a different random process for placing replicas. In the complete BubbleStorm system, it is necessary to maintain replica population despite peer departure. The algorithms which address this issue (not covered here) do not forward replicas from neighbour to neighbour. They (eventually) give each peer a truly independent chance to store a replica. Thus persistent bubbles are not missing exterior edges. However, fading bubbles (and the initial state of managed bubbles) are both affected by this phenomenon.

In BubbleStorm, we opt to solve topological dependency (where it occurs) by paying a bandwidth penalty. When placing persistent replicas with bubblecast, we place a few extra to compensate for the missing exterior edges, a dependency correction factor F.

Over time, our replica maintenance algorithms disperse the replicas more uniformly throughout the network and the correction factor disappears.

The correction factor inflates the number of replicas in the system. Thus, the expected number of search results is no longer λ, but $F\lambda$. These extra results are easy to understand. While the extra interior edges make it hard to find a match, once a match is found they tend to lead you to more matches. If the query lands on a peer with a data replica, there are potentially three further edges which lead to another replica; consider Figure 9.6. This pushes the result distribution from the idealized Poisson to a distribution shifted somewhat to the right. The minimum degree of BubbleStorm peers (16) was chosen, in part, to reduce this extra matches effect.

When copying a replica from neighbour to neighbour, you get a replica tree. Random walks, bubblecast, and flooding all make differently shaped trees, but they are still trees. In a tree with x nodes, there are $x-1$ interior edges, which is $\approx x$ for large x. As each edge is incident on two nodes, the exterior edge count is reduced by $2x$. Thus, the correction factor is easy to calculate using the measurement protocol,

$$\begin{aligned} F &:= \frac{\text{correct exterior edges}}{\text{actual exterior edge}} \\ &= \frac{\sum_{u\in U} w_u \deg u}{\sum_{u\in U} w_u(\deg u - 2)} \\ &= \frac{\sum_{u\in U} \deg u \deg u}{\sum_{u\in U} \deg u(\deg u - 2)} \\ &= \frac{\sum_{u\in U} \deg u^2}{\sum_{u\in U} \deg u^2 - 2\sum_{u\in U} \deg u} \\ &= \frac{D_2}{D_2 - 2D_1} \end{aligned}$$

For a homogeneous network with degree 16, this comes out to $256/224 \approx 1.14$. Thus, the dependency correction factor adds a 14% bandwidth overhead. The traffic weights fed to the bubble balancer consider all traffic costs (including replica maintenance and the dependency correction factor) so this cost is split between query and data bubbles for an overall traffic increase of $\approx 6.9\%$. For a more realistic heterogeneous network, the cost goes down as there are peers with much higher degree and the missing 2 edges play a less significant role.

There is yet a further problem caused by our use of graph edges to emulate random sampling. Within a single bubblecast, it is possible for a node to be reached twice by two independent paths. This, in itself, is not a problem; the bubble balance equations already consider the possibility of placing the same replica into a bin/peer twice. However, once a collision has occurred, further chained collisions are possible. This problem was alluded to in Section 6.2 and is illustrated by the red dashed edges in Figure 9.7.

Each peer has at least 16 neighbours. If it was already reached by bubblecast, then 3 of those neighbours would be bad choices for forwarding. A second incoming bubblecast would pick two edges out of the 15 other available edges. It is $12 * 11/15 * 14 \approx 58.7\%$

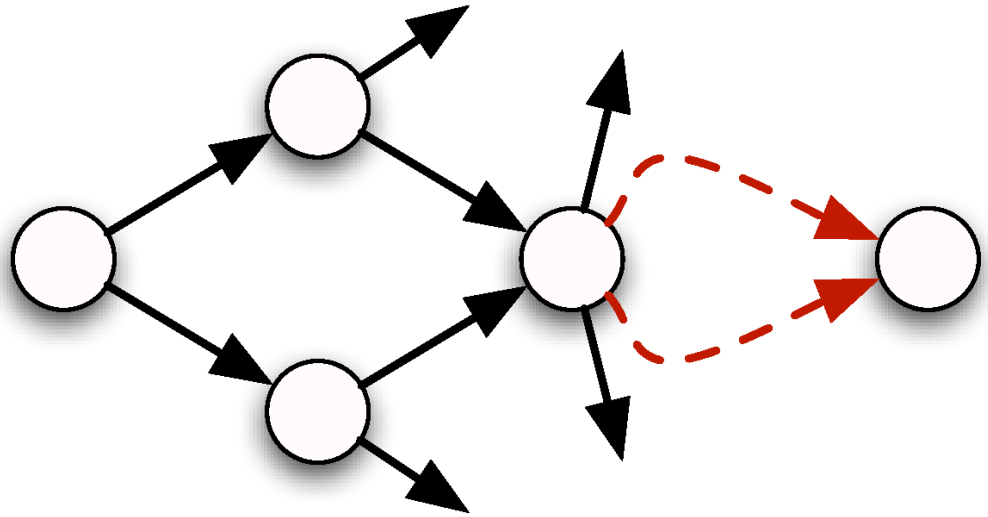


Figure 9.7.: A normal collision in bubblecast (black), as compensated for by Lemma 3, can lead to further chained collisions (red) not captured by our model.

likely that no chained collision occurs. Of course, chained collisions are further reduced by the chance that there is no collision in the first place.

One could imagine another compensation factor, analogous to the F used to amplify the exterior edges from data replicas. However, in our testing this effect has never been very strong. In comparison, the query/data topological dependency had a measurable effect and thus justified the F-scaling. Keep in mind that BubbleStorm already makes a number of safe approximations and rounds up in all other calculations. Due to the Birthday paradox, the balancer tends to avoid saturating the network with a single bubble, as the larger the bubble, the less replicas are placed per message. Furthermore, the balancer uses the approximation g, which overestimates the lost replicas when a bubble becomes large. This means that in exactly the case where chained collisions become likely, the approximation used by the balancer already tends to create extra replicas. Empirically, these other approximations appear to drown out the effect of chained collisions, so much so that in very unbalanced bubbles (where collisions are most likely), the match probability tends to be much higher than it should, rather than lower as one would expect from chained collisions. Therefore, we have elected to ignore this effect in practice.

9.2 Notification

Most of this thesis has ignored the actual delivery of query results. Rendezvous theory ends at the point a rendezvous peer has received both query and data. However, in a real system, the rendezvous peer must deliver the results to the query source. This section details our approach to this vital, if simple, task.

First, one must keep in mind that the BubbleStorm system is designed to aim for λ rendezvous peers in the system. Each of those peers finds that the same document matches the query. However, if the document is large (in our simulations 20KB), it does not make sense to have all rendezvous peers deliver the result, but instead only one.

To ensure the document is sent only once, the notification protocol has four stages. First, a reliable CUSP connection is established. Second, the rendezvous peer reports the unique ID or hash of the document found. Third, the source peer decides to either download or stop the session. Finally, the rendezvous peer sends the result.

Building a CUSP connection requires one round-trip to negotiate the channel encryption and synchronize the sequence numbers used for reliable delivery. Thereafter, the channel goes through slow-start, exponentially increasing the traffic per round-trip until congestion is detected. However, CUSP only builds a single channel between two endpoints. Therefore, if two peers are already connected, perhaps as neighbours, then the existing CUSP connection is reused and this step is skipped. This can have performance implications as negotiation and slow-start add a significant latency penalty.

While executing a query, the source peer maintains a table of the ID/hashes of all the documents found. The first rendezvous peer to report a particular ID is told to send it. All repetitions of the same ID are told to close the connection. The upside to this approach is that it is quite simple to implement. The downside is that if the chosen rendezvous peer crashes while transmitting the result, it is lost. Probably a useful enhancement would be to simply stall the additional rendezvous peers until the first peer completes transmission.

Typically, a query is ended by a timeout. Until that time, results accumulate. The exact latency depends on a number of factors, especially network size/shape, retrieved document size, and network congestion. Nevertheless, results typically arrive in the ballpark of seconds. We have set the default query timeout at 60 seconds, at which point further results are not accepted and still downloading results are cancelled.

While there are several ways we could dynamically estimate how long it would take for a given percentile of results to arrive, we have not yet elected to do so. The interface BubbleStorm provides will report results as soon as they arrive. We think the right approach for a search result list is to just update the GUI immediately, as the results come in. The application can leave the query running until the user closes the window.

Displaying results as they arrive has a few advantages. Mosy significantly, the user gets the first results quickly, increasing the perceived responsiveness of the system. Also, if the result the user seeks appears early, the query can be terminated when the user closes the window. Humans are also good at estimating when progress has ceased; they will see results arrive and as the list changes less and less, they can decide when not many more results are likely to appear.

For an automated system, where the complete result set is desired, our current solution is to just use a large timeout, like 60 seconds or more. If a large timeout is unacceptable, BubbleStorm does not yet offer a solution. Our best suggestion is to include a query with a known number of matches, and gauge the progress by the arrival rate of these matches.

9.3 Queuing

While BubbleStorm seeks to minimize bandwidth consumption and spread load evenly between peers, congestion is an unavoidable fact of life. When it happens, peers need to decide which messages get sent first, and if the congestion persists, which messages will be discarded. Fortunately, the CUSP transport protocol provides BubbleStorm with fairly direct control over these choices.

Briefly, in CUSP, every message is assigned a priority. When a peer has multiple messages ready to be sent, the highest priority message is selected. Then, a packet destined for this host is prepared. CUSP fills the packet up to a full MTU size, selecting messages destined for that host in decreasing priority order. This means that the most urgent message is always sent first, but it may be accompanied by additional messages that do not have the highest global priority, but have the same destination. Keep in mind that packet overhead is 16+20+8+24=48 bytes for Ethernet+IP+UDP+CUSP. By comparison, a query of 20 bytes is quite small. Thus, packing many messages into a single packet can greatly improve network efficiency. However, this not-strictly-priority order can have surprising results.

Picking correct message priorities in BubbleStorm makes a big difference, as we will see in Section 9.4. In decreasing order, BubbleStorm prioritizes topology maintenance traffic first, then gossip/measurement traffic, query response traffic, and finally bubblecast traffic. The reasoning is fairly straight-forward. The topology must remain intact, or the other subsystems will not work. Topology traffic use is also the lowest, and so has highest priority. Measurements are needed to track the state of the network. If measurements stopped flowing, BubbleStorm might get stuck in a state where the system could reduce bandwidth consumption using the balancer, but does not. The measurement traffic cost is also constant, and much less than notification and bubblecast traffic in a congested system. Finally, bubblecast traffic serves to create notification traffic. If notifications are being dropped, there is little point sending a lot of bubblecast traffic to find more. As we will see, prioritizing notification traffic above bubblecast traffic leads to a fairly nice steady state under congestion; high notification traffic leads to a reduction in bubblecast traffic. Reduced bubblecast traffic leads to less results, leading to less notification traffic.

Finally, not all bubblecast messages are created equal. Some messages have a higher counter value; see Figure 9.3. We set the priority of messages with large counter values higher than those with small counter values. Specifically, the priority is the counter value divided by the bubble size. This has two effects: improved latency and graceful decay under congestion.

Prioritized bubblecast improves bubblecast latency, because most rendezvous peers are leafs in the bubblecast tree. To reach these leaf peers, a message must be forwarded from the source over intermediate hops till it reaches its final destination. The latency to the leafs is thus the sum of all these edge latencies. However, peers in the interior of the tree appear on the path to multiple leafs. Therefore, it makes sense to prioritize the early interior messages in the tree, at the expense of slower messages near the leafs of

the tree. Indeed, the bubblecast counter value indicates exactly how many downstream peers are reached.

The more important effect of bubblecast priorities is graceful decay under congestion. When the network is congested, the leafs of the bubblecast trees are dropped first. Because the number of rendezvous peers is roughly proportional to the product of the two bubble sizes, we get better results when both bubble sizes are decreased by the same ratio. By setting the priorities as we have, all bubble types suffer the same relative reduction in size. This maximizes the match probability, despite the inability of the congested system to meet the guaranteed success rate.

9.4 Evaluation

To examine how bubblecast and the complete system behave, this section compares a variety of experiments. The experiments differ in three aspects: peer bandwidth distribution, scenario, and replication algorithm. As in the previous two evaluation sections, we consider two different peer populations, a homogeneous population of 1MBit peers and a heterogeneous population of peers (the same used when simulating the topology; Figure 7.7).

	churn	static	balance
topology parameters			
total peers	20000	1000	1000
online peers	≈ 500-1000	1000	1000
average lifetime	1 hour	infinite	infinite
crash fraction	10 %	n/a	n/a
rendezvous parameters			
query size	20B	20B	20B
document size	20KB	20KB	20KB-20B
document/query ratio	1%	1%	1%
per-peer injection rate (homo)	1/15s	1/60s-60/60s	1/60s
per-peer injection rate (hetero)	1/15s	1/21s-60/21s	1/21s

Figure 9.8.: Simulation scenarios

The scenarios simulated again include the churn scenario with leave, join, and crash events, ala Figure 7.8. We also consider a scenario where the injected traffic of both bubble types increases linearly after the first simulated hour, to explore the system under congestion collapse. Finally, we consider a scenario where only the size of published *documents* decreases linearly after the first simulated hour. This will cause the ratio of S_Q/S_D to change over time and the bubble balance should then react by changing the balance between bubble sizes. The scenario parameters are summarized in Table 9.8.

In all scenarios, documents are published as fading bubbles (Figure 4.1). Queries seek to find documents published between one and five minutes ago. We only query for recent documents because fading bubbles tend to evaporate in the network due to churn

and major topology changing events. Neither managed [51] nor durable [49] bubble types have this restriction, but they are not covered in this thesis. All publish/search events occur with a global Poisson arrival rate per node, irrespective of node degree. Thus, for a homogeneous graph, injected traffic already has the steady-state distribution. However, because injected traffic is a discrete event handled much more quickly than the arrival rate, the distribution is still somewhat bursty. In a heterogeneous network, because the traffic is not injected proportionally to degree, it does not match the steady state. Assuming there is one person using one peer, this usage model is not completely unrealistic.

The three replication algorithms compared are roughly: random walk, bubblecast, and flooding. In order to compare apples to apples, we had to modify the flooding algorithm to be more precise. The flooding algorithm compared here does not use a hop counter, but rather a replica counter like bubblecast. In some sense, the flooding we simulate is actually bubblecast with branch factor equal to degree, instead of two. Similarly, a random walk is just bubblecast with branch factor one. All three algorithms get their bubble sizes from the bubble balancer. Furthermore, all three algorithms prioritize higher count messages. This keeps the three algorithms roughly competitive with each other and allows us to examine the effects on queuing more precisely. To see a less fair, but perhaps more representative comparison with related work, see our earlier work [79] where flooding does not use balanced bubbles and random walks do not leverage heterogeneous peers.

9.4.1 Normal Operation

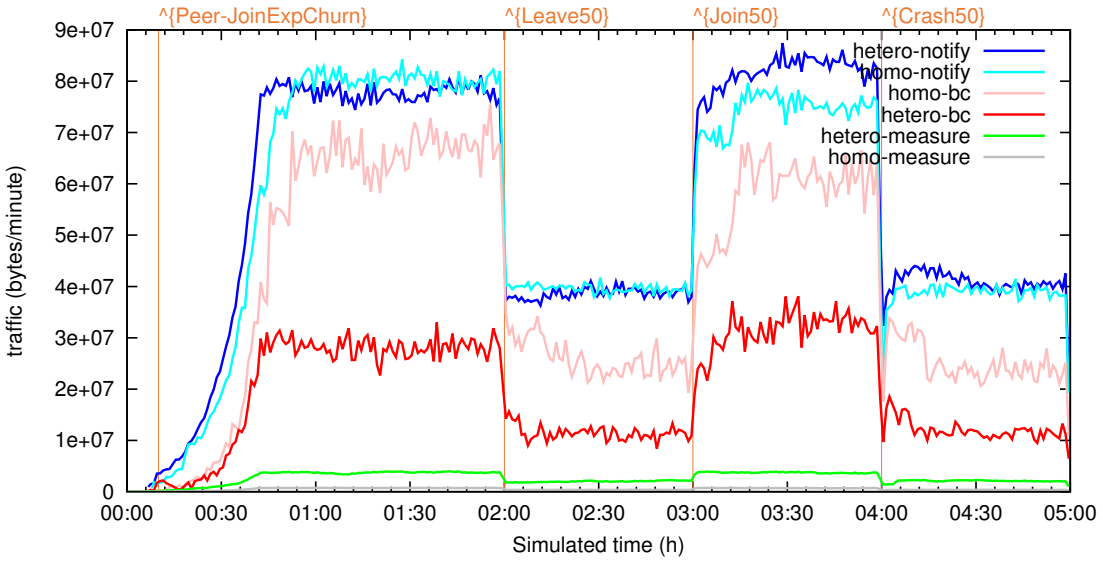


Figure 9.9.: Per-minute traffic breakdown for the churn scenario

To get a feel for what is happening in the churn simulation, consider the per-minute traffic graph. Notice that in Figure 9.9 the bubblecast traffic for the homogeneous net-

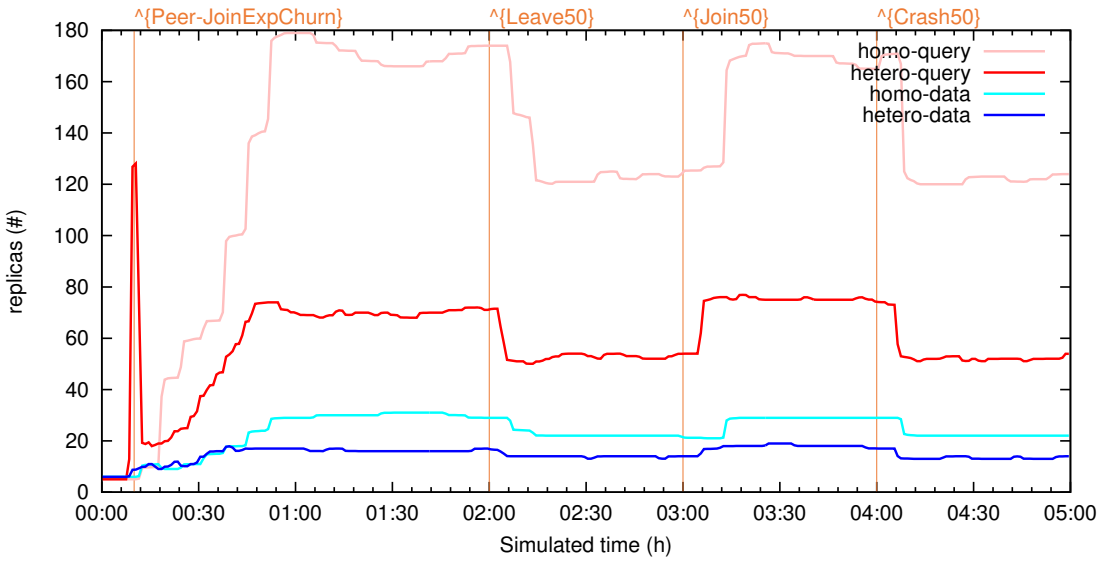


Figure 9.10.: Bubble sizes for churn scenario over time

work is higher. This is simply due to the fact that the injected load for both homogeneous and heterogeneous networks is the same and the bubble balancer exploits the gains from high bandwidth peers to reduce the bubble sizes. The notification traffic in both networks is roughly identical because both issue queries at the same rate. Obviously, the measurement traffic in the heterogeneous network is higher due to the increased number of edges. However, the payoff in reduced bubble size clearly dominates this cost.

The impact of the churn scenario events is also easy to recognize. Notice the sudden jump in homogeneous notification and bubblecast traffic at 3:10, ten minutes after the mass join even. This is due to the time it takes for the measurement protocol to report the sudden change in network structure to the balancer. For homogeneous networks, this takes longer because the measurement protocol runs slower (see Section 8.2). The same delay causes the opposite effect after a massive leave and crash events (4:10); the bandwidth takes a moment to decrease in response to the reduced network size.

That the homogeneous and heterogeneous notification traffic are not exactly equal around 3:30 is not a cause for concern; these two curves correspond to two different simulations. The population fluctuates randomly around the target (see Figure 7.8), due to churn. Furthermore, the heterogeneous network has a mix of various peer capacities and after the join event, that mix can be different than it was before the leave event.

Bubblecast traffic ultimately stems from the bubble balancer. The bubble sizes computed by the balancer from the statistics obtained by the measurement protocol are shown for churn and balance scenarios in Figures 9.10 and 9.11. The churn plot is fairly straight-forward; the bubble sizes are adjusted in response to the population changes, delayed by the measurement protocol.

As for balance, when there are 1000 peers, the query bubble size is ≈ 170 and publish ≈ 30. Comparing traffic, that is $170 \cdot 20 \cdot 0.99 = 3366$ versus $30 \cdot 20000 \cdot 0.01 = 6000$ bytes per bubble. In a balanced system, we would naïvely expect these two to be equal.

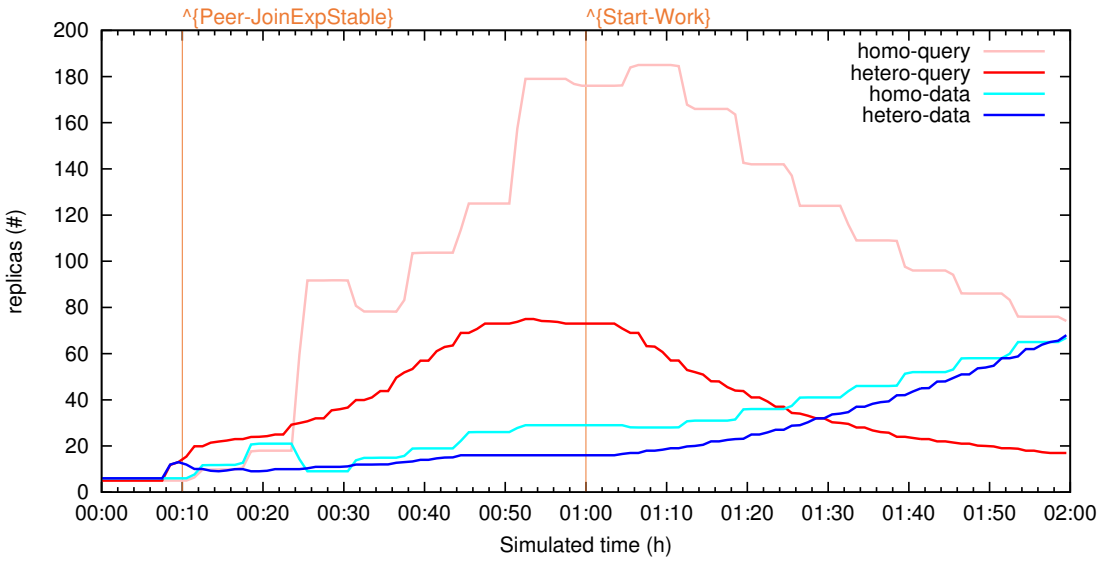


Figure 9.11.: Bubble sizes for balance scenario over time

However, the bubble size 170 is large enough to cause diminishing returns due to peers receiving duplicates. Fortunately, our convex optimizer correctly avoids large inputs to g and opts for a smaller query bubble size. This also explains why, when the network is reduced in size, it is the query bubble that shrinks much more than the data bubble.

The bubble sizes in the balance scenario (Figure 9.11) are much more interesting. In this simulation, we had set document sizes to start decreasing after one hour. This means that S_D decreases and the optimal trade-off between query and publish bubble sizes shifts. Thus, every time the mesurement protocol terminates, from 1:00 to 2:00, the system calculates a new trade-off. As expected, the data bubble size grows, as each data replica now costs less traffic to produce. Meanwhile, the query bubble size shrinks as the larger data bubbles mean that query bubbles need not be so large. This same rebalancing effect would also occur if the injected publication *rate* were to change instead of the injected document *size*; both factors affect S_D.

To smooth out the measured traffic over time, BubbleStorm uses an exponential moving average. Peers report their traffic as 20% of their traffic since the last measurement and 80% of the last reported value. The averaging is applied *before* the local value is fed to the measurement protocol. Thus, the system can still respond immediately to large-scale topology events. If half the nodes crash, the measurement protocol will report half the traffic when the subsequent measurement completes. By design, the exponential moving average dampens the reaction of the system to changes in the ratio of S_D/S_Q, preventing large swings in response to temporary changes in traffic mix. However, this also means it slows the reaction of the system to the balance-changing simulation.

For the real-world, this moving average makes sense. We want to respond quickly to network outages, but we don't want transient fluctuations in the traffic distribution to cause radical changes in bubble balance. For persistent bubbles in particular, a changing bubble size requires maintenance traffic [49, 51]. However, in this experiment, it means

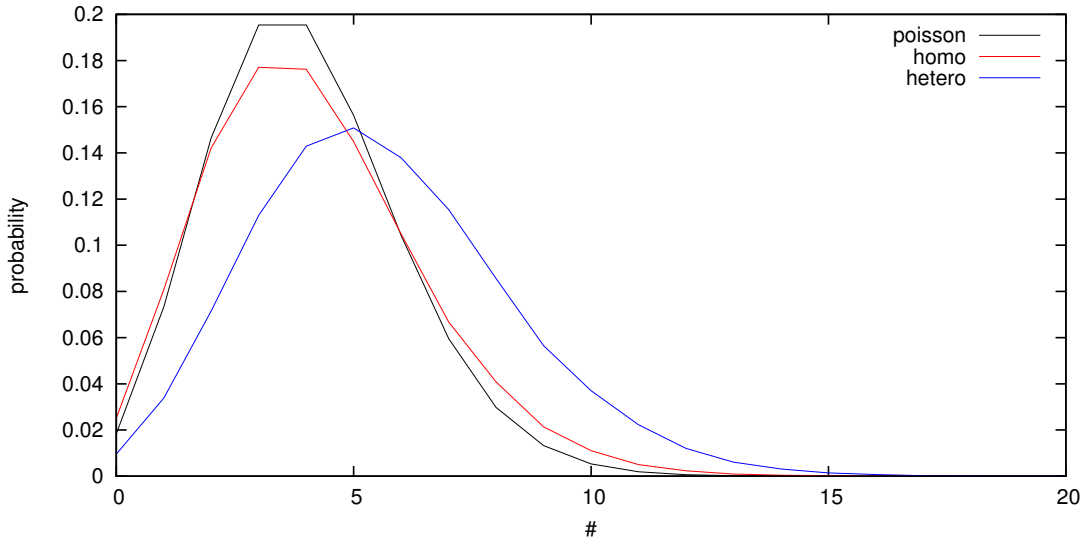


Figure 9.12.: Probability distribution of search results for λ=4

that the changing document size affects heterogeneous networks faster. This is because the measurement rounds complete more swiftly. In turn, this causes the old value to be multiplied by 80% more frequently, and so the exponential moving average tracks the changing traffic ratio more closely.

Thus, the faster heterogeneous measurement round switches make the heterogeneous curves in Figure 9.11 smoother due to smaller and more frequent changes. Furthermore, their balance shifts more quickly, nearly keeping pace with the actual bandwidth distribution. As a future enhancement, it might make sense to weight the exponential moving average according to the time between measurement rounds. This would cause both homogenous and heterogeneous networks to track the traffic ratio at the same speed.

Now, we turn our attention to BubbleStorm's correctness. Recall from Theorem 4 that we expect BubbleStorm to return query results according to the Poisson distribution. However, as discussed in Section 9.1, since we used the same network to forward both queries and documents, there is a dependency not captured by the idealized analytic result. For a homogeneous network, we argued that it was necessary to amplify the results by a factor of 1.14 in order to correctly bound the failure rate. Furthermore, we also expect that the approximation made in Theorem 7 will cause us to overestimate the necessary bubble sizes when the network is highly heterogeneous.

To check how closely theory meets practice, we recorded the number of document replicas found per query over the period from hour one to two in the churn scenario. During this time, the network is quite stable, though it is subject to crash events. The results for bubblecast are shown in Figure 9.12.

Firstly, the probably of zero hits for the homogeneous network is slightly higher than predicted for $\lambda = 4$. I believe there are enough samples in this plot that this is not measurement error, but a true defect in the success rate. This difference disappears when crash events are removed, so it is most likely due to lost messages. One of the key

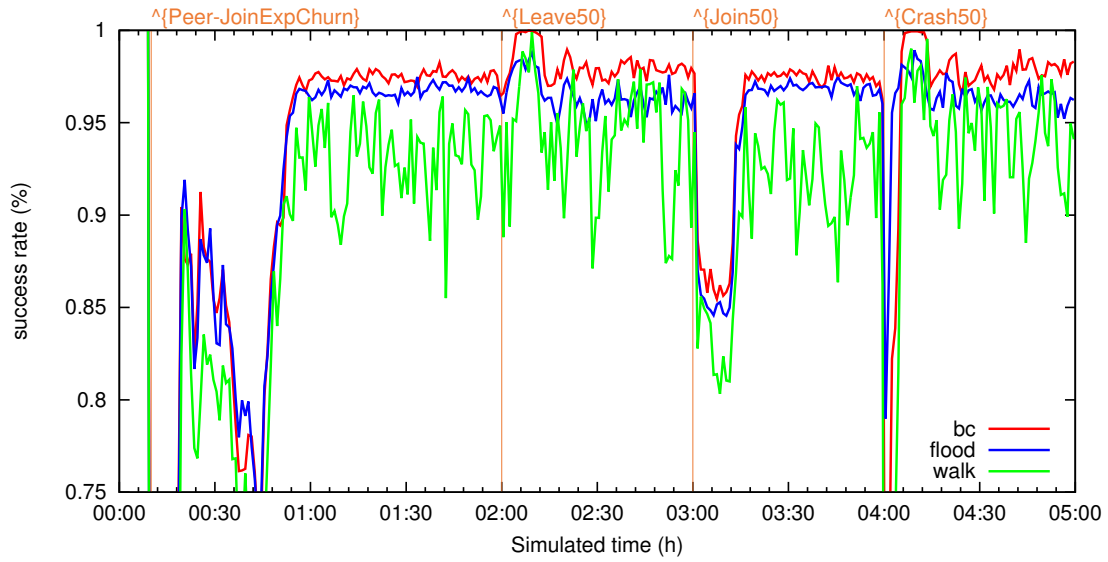


Figure 9.13.: Query success rates under churn in a homogeneous network

advantages of BubbleStorm is that one can tune λ to deal with the expected worst-case network events. As long as one keeps in mind that λ is for a network without losses, then BubbleStorm appears to be working exactly as intended.

Next, except for 0, the mass of the homogeneous result distribution is shifted slightly to the right when compared with the idealized Poisson distribution (though not as far as the heterogeneous result). This corresponds to the dependency compensation factor $F = 1.14$. Recall that this factor corrects the mass at zero, but increases the average number of results by 14%. The plot confirms this behaviour; the curves align at zero, but the average is 14% right-shifted.

Finally, consider the heterogeneous topology. We have a rather small network of 1000 peers with capacity ranging from degree 16 to 1280. We are clearly in the realm where a few peers completely dominate the network. Thus, the approximation of using w_m to simplify the balance equation will clearly cost us some unnecessary traffic. Indeed, the network is returning 5.6 matches instead of 4, a significant shift of mass to the right. However, we already know from Figure 9.9 that despite this wasted effort, the bandwidth is still significantly decreased by leveraging heterogeneity. Furthermore, we know from Theorem 8 (optimality of BubbleStorm) and Figure 5.6 (heterogeneous bubble sizes vs. network size) that if the network were to grow with this same bandwidth distribution, the wasted traffic would decrease. So, BubbleStorm should perform even better at scale, though simulating networks large enough to see this effect is beyond the capabilities of our lab.

We now further examine the rate of successful queries. While the results per query are interesting, the main concern is the failure probability. To that end, consider Figures 9.13 and 9.14. In these plots, the success rates of the three replication algorithms are compared as they are subjected to the churn scenario's events.

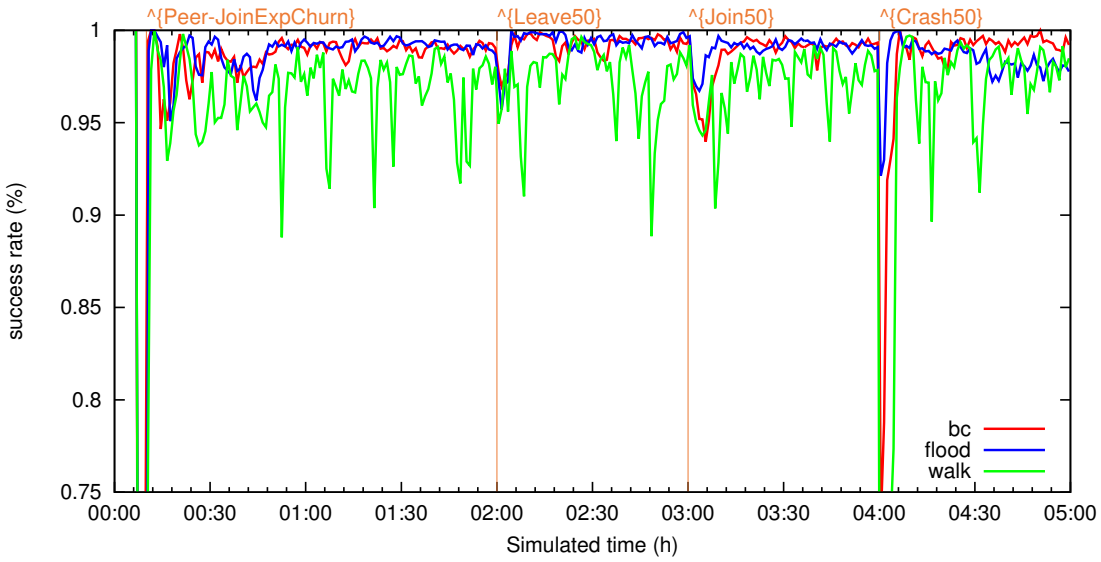


Figure 9.14.: Query success rates under churn in a heterogeneous network

Right away, it is clear that random walks are a poor choice. As argued in Section 9, a lost packet affects a random walk much more significantly than a tree. It's also quite clear that the slower convergence times of the measurement protocol on homogeneous networks makes topology changes affect the success rate for much longer. On the join event (and the initial exponential join), the success rate is too low as the bubbles need to be scaled up. On leave events, the success rate is temporarily too high as the bubbles need to be scaled down in size. On crash events, success initially plummets due to lost messages, but thereafter is too high as the bubble size needs to come down. These effects are also visible on the heterogeneous network in Figure 9.14, though they are less pronounced.

In heterogeneous networks, the success rate is higher than it should be. However, this is to be expected since we already saw in Figure 9.12 that our heterogeneous approximation led to more results than requested. Still, it is quite instructive to see how much a difference a slightly increased effective λ has on the success rate. Both plots have the same scale, and the difference is quite noticeable.

Finally, the perhaps most surprising result is that flooding performs markedly worse than bubblecast on homogeneous networks, but does not suffer significantly on heterogeneous networks. This is due to the collision chaining effect illustrated in Figure 9.7. For a flood-based replication approach, a single collision is magnified to 14 more collisions in a homogeneous network. In a heterogeneous network, the query size is roughly the same as the average degree. Thus there is little opportunity for chaining to appear.

We argued that the collision chaining effect is normally small enough to be neglected, overwhelmed by the approximation g. However, that argument presupposed branch factor 2 bubblecast and degree 16 peers. Indeed, we specifically chose those values to ensure that collision chaining does not matter.

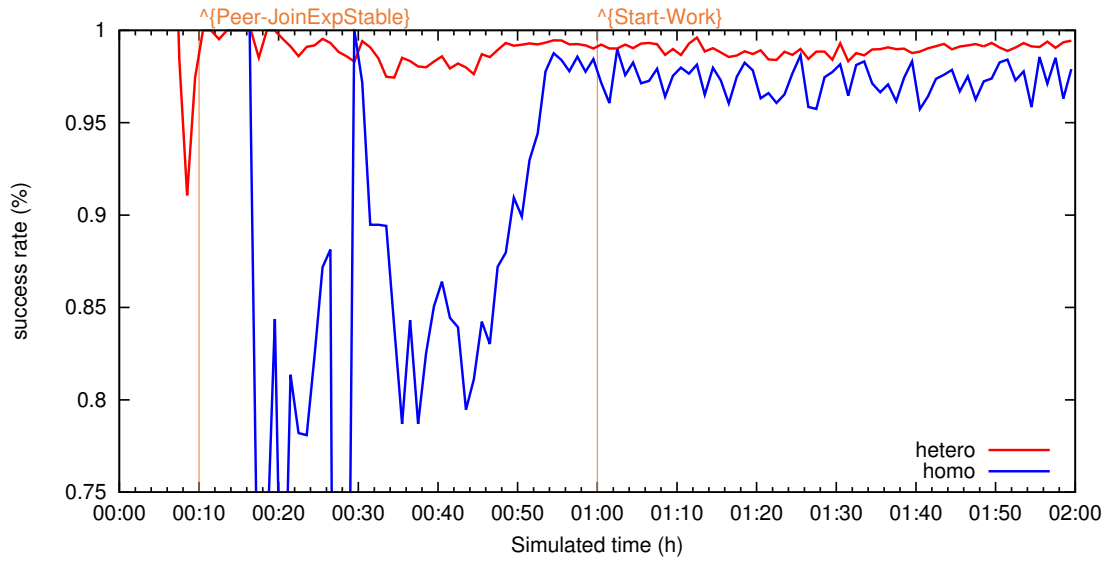


Figure 9.15.: Query success rates under changing bubble balance

Finally, Figure 9.15 shows the effect of changing bubble balance. First of all, the heterogeneous success rate is again higher than it should be, due to our approximation using w_m. The heterogeneous plot stays roughly where it should, but the homogeneous plot has periodic dips. In fact, each of those dips corresponds to a new measurement round. The publish bubble size increases after each measurement, but fading bubbles take no corrective action. Since the queries look for data that is potentially 5 minutes old, they sometimes match against publish bubble that are too small. Still, the average hovers around the target 98%, so the system is performing correctly. The balancer does a good job finding bubble sizes as the ratio of S_D to S_Q changes for both hetero- and homogeneous networks, confirming that the theory matches practice.

9.4.2 Homogeneous Congestion Collapse

So far, we have seen that BubbleStorm performs well even in the face of major topology changes. However, to truly understand a system, one must break it! In this section, we subject BubbleStorm to a true torture test. The static scenario increases the traffic linearly after the one hour mark, ending with a 60-fold increase in traffic. This pushes the traffic requirements beyond what the network can bear, and we will see that this tells us a lot about the system.

Figure 9.16 shows the average UDP traffic sent by each peer. At the one hour mark, both query and publish events increase linearly in frequency, so unsurprisingly, the traffic shoots up. All our peers in this network have 1Mbs DSL, which has only 128Kbs = 16KBs uplink capacity. The simulation includes Ethernet and IP traffic overhead not measured here, so the plot shows we are pretty close to completely saturating the link by the end of the hour.

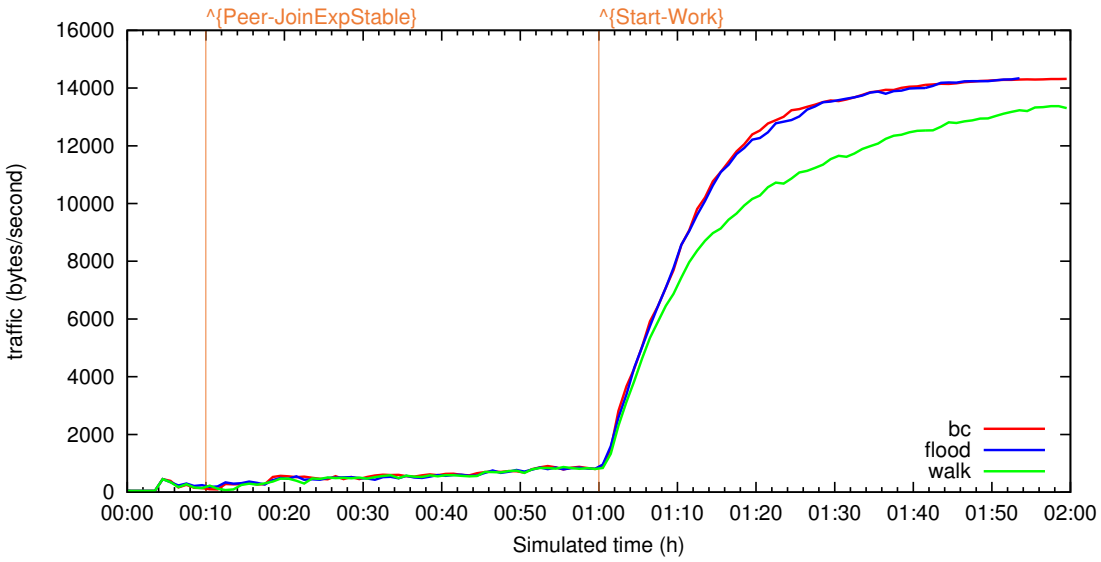


Figure 9.16.: Average uplink traffic per peer in the homogeneous static scenario

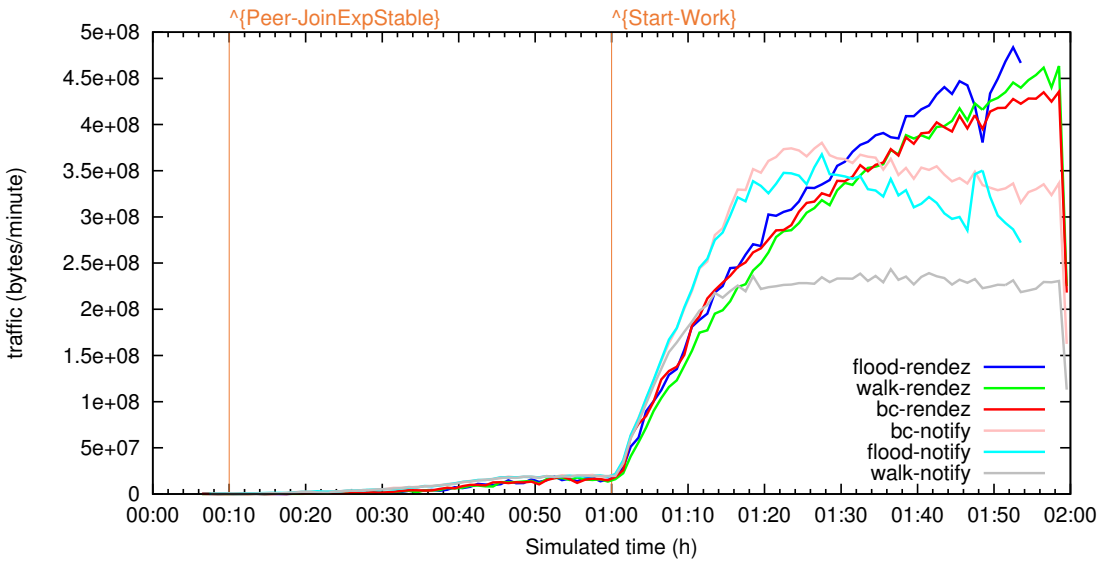


Figure 9.17.: Traffic breakdown for the homogeneous static scenario

Once congestion sets in, messages must be dropped. However, because we use CUSP priorities, not all traffic is affected equally. For example, no gossip or topology messages are lost. As designed and Figure 9.17 shows, rendezvous (bubblecast/flood/walk) traffic begins to get dropped when congestion sets in. Decreased rendezvous traffic results in less matches, which in turn leads to less notification traffic. This nice feedback loop results in us getting results back to the user with a reduced λ, which is pretty much the best we could hope for. The notification (search result) traffic alone exceeds the network capacity, so no system could guarantee exhaustive search under these conditions.

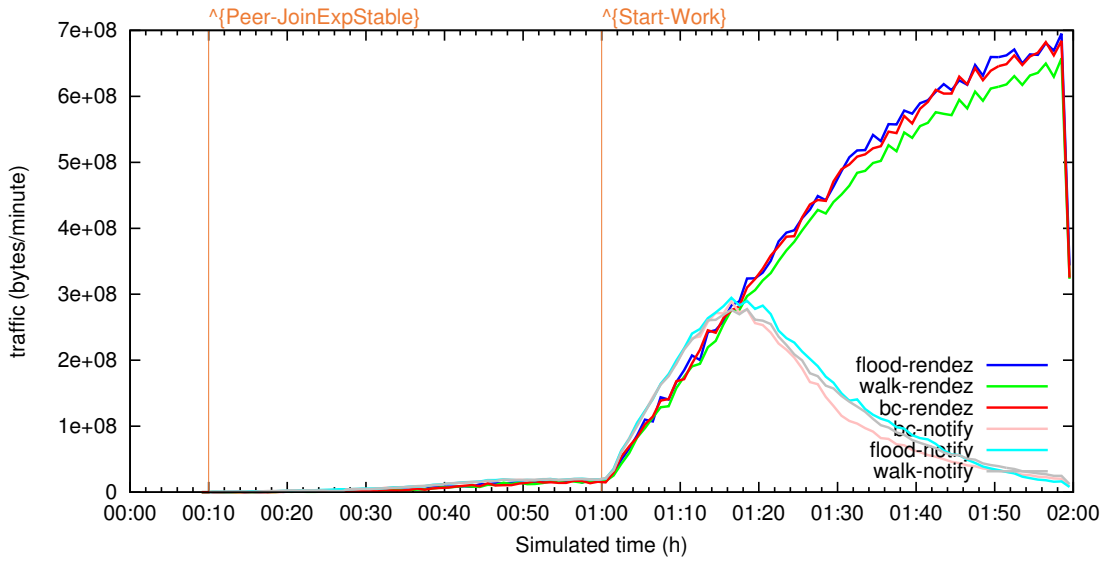


Figure 9.18.: Traffic breakdown for a buggy homogeneous static scenario

By comparison, consider the buggy network in Figure 9.18 where rendezvous traffic has higher priority than notification traffic. There, the rendezvous traffic completely eliminates the notification traffic from the system. Naturally, this means that there are no results being returned to the user, and all of the system's effort is in vain.

When the rendezvous traffic decreases, this means that bubbles are smaller than they should be. Because we prioritize larger counter values, the losses tend to be on the leafs of publish/search trees. Furthermore, though not shown here, both query and publish bubbles are affected equally. This is exactly the trade-off we want according to rendezvous theory. The improvement to search success decreases as bubbles get larger. Thus, by reducing all bubbles equally, success is maximized despite losses. This is confirmed in Figure 9.19. There we see the success rate begins to fall around 1:15 and reaches 50% success rate around 1:30. Keep in mind that workload is increasing linearly, which means that request rate at 1:30 is double that at 1:15. Since the system is at maximum capacity, it cannot return any more results. Thus, getting results for half of the issued requests is pretty much the ideal result.

While the curves for bubblecast show great promise, we can see that the alternative replication algorithms run into problems. Random walks, as ever, suffer badly in the face of packet drops. Even light congestion can cause drops and thus bubbles are much smaller than they could be. Indeed, random walks even failed to fully saturate the system as Figure 9.16 shows.

The situation for flooding is less dire. However, in a homogeneous network, "flooding" is just bubblecast with branch factor 16. Thus, we actually expect it to perform fairly well; our concerns with flooding had to do with hop counters (which we replaced with a replica counter in this test) and poor load balance due to a dependency on degree. Since load is injected uniformly at random in the graph, and all nodes have the same degree, flooding should not suffer from poor load balance in this test either.

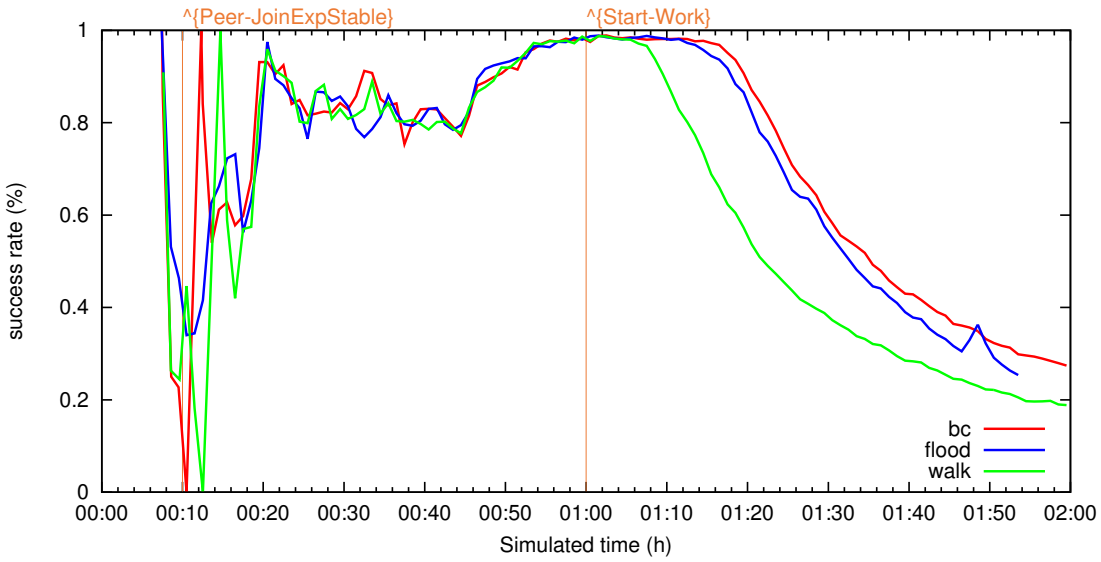


Figure 9.19.: Query success in a homogeneous network under increasing load

The only real benefit bubblecast has compared to flooding in this simulation is that bubblecast has a wider variety of counter values. Branch factor 16 reduces the bubblecast priorities very quickly. As a consequence, it is harder to reduce all bubbles equally. Many/most of the counter values under flooding are 1. When a peer must drop traffic, it can't distinguish between these messages. Thus, some bubbles get lucky and suffer very little reduction, while some bubbles are unlucky and lose nearly all their leafs. As argued earlier, rendezvous performs best when all bubbles are reduced equally. As this is harder to achieve with flooding, it obtains fewer results, despite the higher rendezvous traffic shown in Figure 9.17. Also, the flooding experiment had to be terminated early as it incurred so much queueing that the simulation system ran out of memory.

Finally, we turn our attention to latencies in Figure 9.20. We consider two latency measures; time until a rendezvous peer identifies the match (raw) and time until the requesting peer has received it (rtt). Obviously, random walks perform poorly for both, and it only gets worse as congestion sets in.

Before the system is loaded, the notification latency dwarfs the rendezvous latency for both bubblecast and flooding. If the documents were smaller, it is possible the shorter paths of flooding would be of some benefit. However, as we've seen, large branch factor has penalties in correctness (due to chained collisions) and performance under congestion (less variety in counter values). Still, it might be worth trying bubblecast with branch factor 4 (versus 16), to reap most of the latency benefit and (perhaps) suffer little of the penalties.

Once congestion sets in, things get more interesting. Both bubblecast and flooding rendezvous latency shoot up. Keep in mind that due to our prioritization, most of the latency is incurred in the last few hops. Thus, the shorter forwarding path for flooding benefits it much less than one might expect. As the load gets very high, something very unexpected starts to happen: bubblecast performance begins to improve! There is a

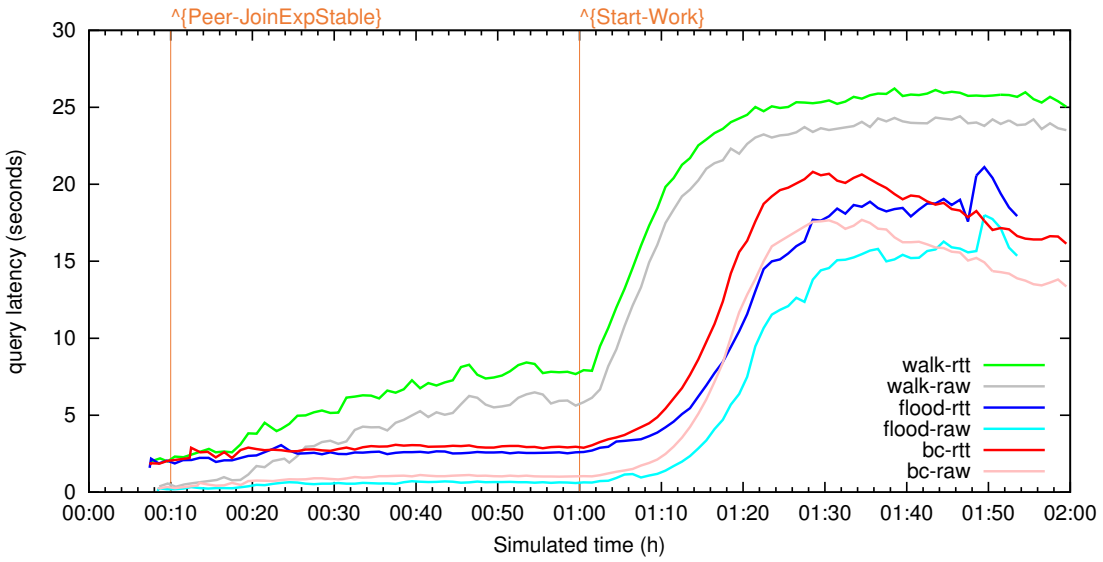


Figure 9.20.: Search latencies in a homogeneous network under increasing load

fairly simple explanation: less results are being found because leafs on the search tree are being pruned. Thus, a larger fraction of the *successful* results are being found in the interior of the tree. We don't see this benefit for flooding, because the rapid decrease in counters means that most of the matching is still happening when the counter is 1, which have the lowest priority.

9.4.3 Heterogeneous Congestion Collapse

For the heterogeneous overload test, the network has higher capacity due to the larger peers. Thus, we need to increase the frequency of queries and publishes to saturate it. Furthermore, as we can expect from Figure 9.9, rendezvous traffic will take a backseat compared to notification traffic. Indeed, Figure 9.21 bears this out; notifications consume the majority of traffic. We can already anticipate that things will not turn out well in this scenario. BubbleStorm optimizes for rendezvous traffic, not notification traffic.

The next sign that something is amiss comes in Figure 9.22. Here we see that after 1:43 the average time for bubblecast to match on a rendezvous peer (bc-raw) is more than the average time for results to be delivered to the query source (bc-rtt). This seems impossible, because for a given result, the rtt must be larger than the raw time. However, these plots only consider *successful* results. Therefore, we can conclude that there are a large number of results found by the rendezvous system which fail to get delivered to the query source. Furthermore, it must be exactly the uncongested rendezvous peers which manage to deliver their results.

Recall that we use degree proportional to capacity. According to rendezvous theory, a double capacity peer can match four times as many results. That peer receives twice as many queries q and twice as many publishes p. From the point-of-view of processing the received load, this isn't typically a problem. The black box model doesn't apply at a

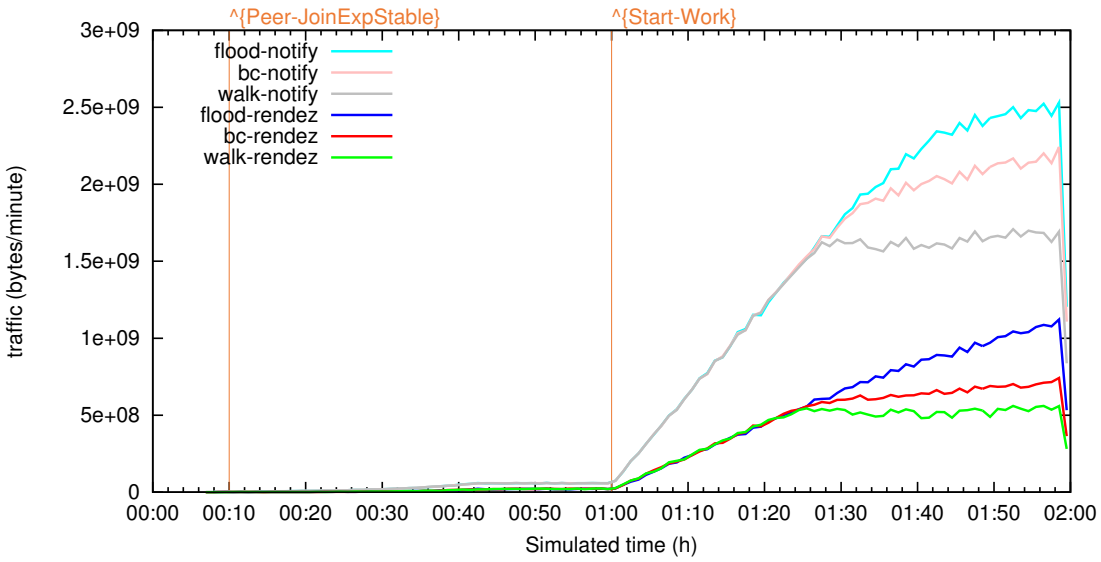


Figure 9.21.: Traffic breakdown for the heterogeneous static scenario

peer; the peer is free to use an index or some other fancy data structure. For a database table with a BTree index, the cost to process all the queries might simply be $O(q \log p)$. Similarly, storing the publications costs just $O(p \log p)$, quite reasonable.

The problem that appears in this scenario, however, is that the traffic to report these matches grows as $O(qp)$. That means that a double degree peer needs to send four times as many notification messages. Since we set degree proportional to bandwidth, it should now be clear why the system is not taking this well. Rendezvous traffic is well balanced, proportional to capacity, but notification traffic grows with the square.

As a result of this effect, the highest capacity peers are overloaded around the 1:25 mark. As the traffic continues to grow, more and more peers become overloaded. This effect explains why the notification bandwidth in Figure 9.21 continues to increase after the effects of congestion begin to be felt; there is as-yet unused capacity on the *lower* capacity peers. The system has poor *notification* load balance.

This effect also appears in the success rates shown in Figure 9.23. In this plot, we also measure the success rate of the rendezvous step alone (raw), as well as the success rate including notification delivery (rtt). In the homogeneous case (Figure 9.19) we did not plot these separately, because they had the same curve. It is easy to see that in the heterogeneous case, bubblecast finds many matches that the notification subsystem is then unable to deliver. This is because the majority of results are found on high capacity peers which are completely overloaded and unable to deliver the result notification.

Curiously, the poor rendezvous load balance of flooding actually serves to improve performance in this scenario. Recall that we expect flooding to place undue rendezvous load on high capacity peers. They will send replicas proportional to the square of their capacity (Section 9), which is bad for rendezvous traffic balance. However, rendezvous traffic is not the dominant traffic in this scenario. Consider that every message has both a receiver and a sender. If high capacity peers send too many replicas under flooding,

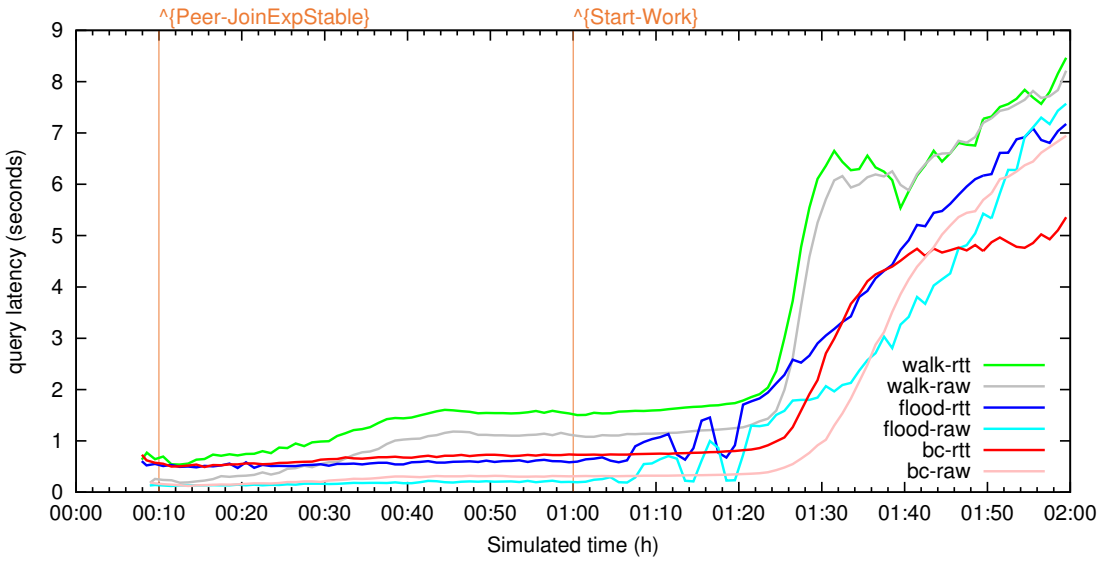


Figure 9.22.: Search latencies in a heterogeneous network under increasing load

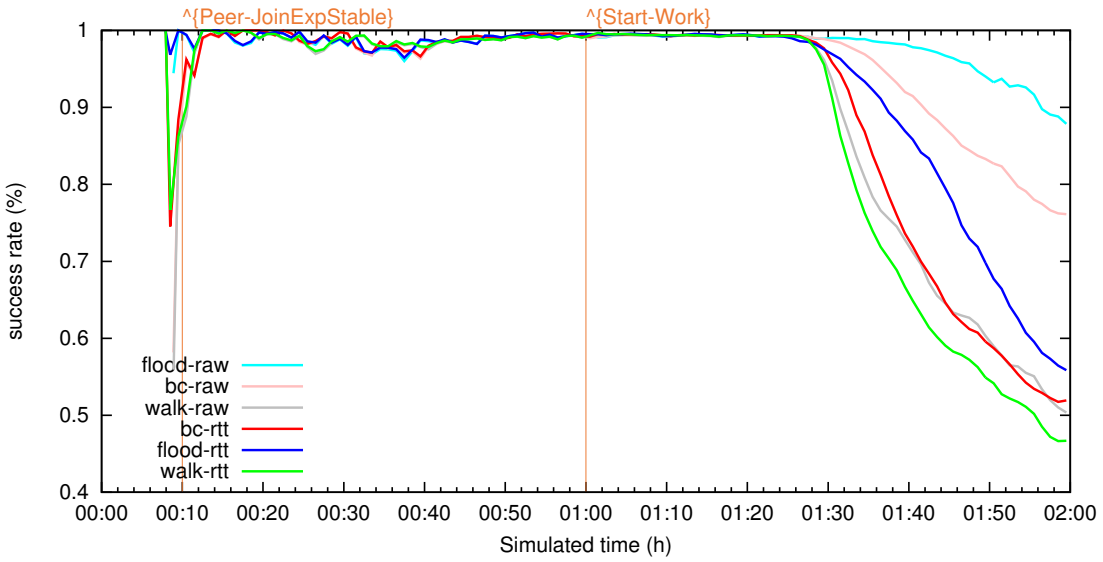


Figure 9.23.: Static query success rates in a heterogeneous network

they must *receive too few*. Thus, due to this poor rendezvous load balance, the system matches less on high capacity peers than designed. This serves to improve the load balance for notifications, because high capacity peers (the bottleneck) are underutilized.

Despite the negative effect on notification traffic, the choice to set degree proportional to bandwidth is *not* wrong for rendezvous traffic. We know from rendezvous theory that this choice yields the most matches per invested unit of rendezvous traffic. If anything, these results show the low cost of running BubbleStorm to solve the rendezvous problem on a real system. The result set, which must be delivered by any correct system (black

box, key-based, or otherwise), dominates the traffic consumption. For small networks, then, it might make sense to set degree proportional to the square-root of bandwidth. This way, the notification traffic would be proportional to capacity.

On the other hand, BubbleStorm is designed primarily to deal with very large networks. In our simulation, there are only 1000 nodes due to memory limitations. However, in a large system with 10 million nodes, the bubble sizes would need to scale up 100 times higher[1]. At this point, bubblecast traffic would dominate notification traffic, and one would again want degree directly proportional to bandwidth.

On yet another hand, in a real system, there will likely be more than one result per search, increasing the notification load further. However, we have always intended to build a Top-K query algorithm into BubbleStorm, similar to [7]. This would again keep the notification costs down.

It's hard to imagine a good static trade-off between these two types of traffic, because their relationship depends heavily on the application, workload, network topology, and size. Due to the myriad of factors involved, it might make sense for a future version of BubbleStorm to measure notification traffic in addition to bubblecast traffic. Then, a smart algorithm could decide where in the spectrum between square-root and linear degree peers should position themselves.

This thesis leaves the solution to the notification versus rendezvous traffic problem open. Our current position is that BubbleStorm needs to perform well when the network is large. Small networks are easy. Leaving degree proportional to bandwidth means that the system behaves best where the problem is hardest. However, a future adaptive approach could capture the best of both worlds.

[1] It will actually be much less because the w_m approximation become sharper and the balancer will have the freedom to more closely match query and publish bubble costs. I'd estimate just a 30- to 50-fold increase for this application setup at 10 million nodes.

10 Outlook

BubbleStorm started with lofty goals and matured into a complex software engineering project. Now that we have a concrete system, it is natural to ask how close to our goals we came. The primary goal of BubbleStorm was to bridge the gap in usability between peer-to-peer systems and databases. We consider in this Section what worked out well, the progress made, issues remaining, and give an outlook towards future work.

The main non-issue is bandwidth. Some researchers confronted with BubbleStorm mumble something about square-root bandwidth cost and then dismiss it out of hand. We believe that this issue is a complete red herring. The area where BubbleStorm is *particularly* strong is networking. It has excellent bandwidth, latency, load balance, fault tolerance, and structure.

We have shown that the black-box matching problem fundamentally requires square-root traffic complexity. So long as one intends to build a system with capabilities approaching those of a database, one must be prepared to pay this cost. The "competition", in the form of key-based systems, only *appears* to offer logarithmic costs. The moment they need to build a more sophisticated matching algorithm, their bandwidth advantage is greatly diminished, often with an additional penalty to latency.

However, in the BubbleStorm project, we are not opposed to the special-case handling of simple queries. In fact, the higher layers of BubbleStorm already offer constant-bandwidth key-based lookups [49]. That said, when building a complete solution, one cannot simply cherry pick the easy problems. The system must be prepared to handle complicated queries, while still offering optimization opportunities. This is the approach we have taken with BubbleStorm.

Furthermore, bandwidth is constantly improving. When the project began, perhaps one could argue that square-root traffic costs were too high. Anecdotally, from when BubbleStorm project started to the day this thesis was published, my bandwidth at home increased 50-fold. The increase in global population (and thus potential query workload) has barely increased over this time frame. Every day it makes more and more sense to trade bandwidth for latency and query power.

The next area where BubbleStorm is particularly strong is latency. Certainly, one can do no better than logarithmic routing depth, so long as the memory consumption of peers is bounded. While bubblecast only forwards searches with a branch factor of two, we have seen that the time to negotiate a connection and send the payload from a responder to the query source overwhelms the bubblecast cost. The only area where BubbleStorm's latency might be improved would be through the exploitation of network locality. For example, a friend-to-friend overlay might be used to further improve the latency of intermediate hops.

On the load balance front, we have shown that BubbleStorm achieves the best steady-state rendezvous workload distribution possible. However, Section 9.4.3 also showed

that notification traffic creates hot spots on high capacity peers. It would be interesting to find a way to balance these two types of traffic, despite their fundamentally different frequencies. This is a self-contained and probably easy-to-solve problem. Compared to structured systems, the situation is vastly simpler. As those systems partition based on key, popular keys will be fundamentally overloaded, particular if access is Zipf distributed (like keywords). While there are approaches [2, 12, 88] to solve this, the problem must be considered anew for each layer of complexity added as, inevitably, more powerful queries are needed.

Thanks to its careful design, BubbleStorm can tolerate extreme network catastrophes. Half of the population can simultaneously join, leave, or crash and the system stays intact. While these large scale events do have a temporary impact on the correctness guaranteed by BubbleStorm, the system returns quickly to normal operation. Typically the system is fully healed after a single measurement round detects that corrective action is required. Intermittent failures due to churn and individually crashing peers poses no serious threat to correct operation. Even under extreme churn, the application developer can simply increase λ to cope with the anticipated worst-case scenario. Furthermore, due to the unstructured nature of BubbleStorm, no particular network connections are required and the system works fine even when faced with the real-world's incomplete network connectivity.

The unstructured nature of BubbleStorm has not proven much of an obstacle. While structured systems are designed to exploit particular network topology, BubbleStorm demonstrates that this dependency is mostly unnecessary. It is quite possible to build an efficient rendezvous system (and key-based routing) on an unstructured system.

One of the key contributions of this thesis was the development of rendezvous theory using the Poisson formulation. Convex optimization, combined with a few carefully chosen approximations, yielded a system which depends only on graph expansion. This work is what enables BubbleStorm to guarantee performance and correctness on unstructured systems, all while balancing bubbles for asymptotically optimal traffic consumption. Furthermore, rendezvous theory empowers us to leverage the previously untapped capacity of heterogeneous peers. This unique feature of BubbleStorm is especially important, as we have shown that heterogeneity pays off in the square!

Unfortunately, while BubbleStorm marks significant progress, it does not yet reach our database-parity goals. In particular, BubbleStorm does not offer atomic commits, unique/foreign key constraints, or joins. Atomic commits could probably be supported using a two or three phase commit protocol on top of the per-document sequencing offered for maintained and durable bubbles [49], similar to the approaches taken by Ivy [61] and GauthierDickey [31]. However, this would incur significant costs and has not been investigated yet. Similarly, key constraints could probably be layered above the replication algorithms. We have had some thoughts about how to handle joins, as discussed in [50]. However, all of these issues are very complicated future work. BubbleStorm at present does not deliver these important features.

Finally, one of the major goals of BubbleStorm was to make peer-to-peer accessible to network non-experts. The black-box and bubble abstractions do an excellent job of sheltering users from routing concerns. During development, we ran several short

lab courses where students used BubbleStorm, and our anecdotal experience was that students were perhaps too well insulated from networking, and needed prodding to actually learn how and why their projects "just worked".

BubbleStorm was developed as a research project, not for commercial deployment. Thus, we made a few design decisions that would need to be revisited before BubbleStorm could be delivered to the average developer. First, we implemented BubbleStorm in Standard ML. While this was a good choice in terms of getting the job done and learning new things, it is not a language familiar to most people. This limits who can contribute to the BubbleStorm core codebase directly. Furthermore, while BubbleStorm does offer an interface to C/C++/Java, this interface is fairly complex and not sufficiently well documented. Finally, while CUSP does take steps to solve the NAT problem, until a solution like the one proposed at the end of Section 7.2.1 is deployed, BubbleStorm (like all peer-to-peer software) will be difficult for end users to configure. Nevertheless, despite these minor issues, the BubbleStorm system is certainly mature enough to support a sufficiently motivated developer.

In conclusion, BubbleStorm goes a long way towards building a complete system that could be used by the network-unsavvy community at large. It certainly goes further than most research projects, having a solid theoretical underpinning, clear vision, well engineered design, and mature implementation. There remain several interesting open research topics (atomicity, joins, etc.), which a motivated researcher could pursue to achieve database parity. Finally, BubbleStorm is a near-optimal solution to the rendezvous problem, and provides a firm foundation on which future work can be built.

Unfortunately, the research community appears to have lost its taste for peer-to-peer, moving on before most of the technologies were ready to be commercialized. In my opinion, the popularity of structured key-based routing was partly to blame. These approaches were always a dead end for building a complete system, yet they were superficially elegant enough to distract the majority of research. Sadly, today it seems we will be left with cloud services, which have the paid developers needed to develop mature systems. Peer-to-peer's theoretical advantage, no centralized authority, also appears to have been its downfall, no centralized development.

A Notation and Variables

Symbol	Defined	Description
c	2.2	The number of columns in the grid formulation
C_u	2.1	The link capacity of a peer (in bytes/second)
$d \in D$	2	A particular replica of stored data
D	2	The set of all data stored by a given workload
$D_i := \sum_{u \in U} C_u^i$	8	The degree sum of peer capacities
$\mathbf{E}(X)$		The expected value of random variable X
f		A generic function used with many definitions
$g(z) = 1 - e^{-z}$	2.3.1	A safe approximation for the sum of unlikely events
G	6.1	The adjacency matrix of a network graph
i, j		Generic index variables used with many definitions
I_X		A generic indicator random variable; X=1 when X true
λ	2.3	The parameter which controls $\mathbf{P}(M = 0) \leq e^{-\lambda}$
λ_2	6.1	The second eigenvalue of the graph topology
M	2.3	The number of peers which receive both q and d
$M_{aggregate}(R)$	2.1	The total traffic for a given rendezvous algorithm
$M_{bottleneck}(R)$	2.1	The utilization of the most loaded peer
$M_{expected}(R)$	2.1	The highest expected utilization amongst all peers
$n = \|U\|$	2.1	The total number of peers participating in the network
$\mathbf{P}(X)$		The probability of event X
$q \in Q$	2	A particular replica of a query
Q	2	The set of all queries processed in a given workload
r	2.2	The number of rows in the grid formulation
$R(d) \subseteq U$	2	The set of peers which receive a replica of d
S_d	2.1	The size (in bytes) of a particular piece of data
S_D	2.1	The size (in bytes) of unique data in the system
S_q	2.1	The size (in bytes) of a particular query
S_Q	2.1	The size (in bytes) of all unique queries processed
T	5	The set of bubble types
$u \in U$	2	A particular peer in the network
U	2	The set of all peers in the network
V	6.1	The diagonal matrix with peer degrees
$w_u = C_u / \sum_v C_v$	2.5	The chance that a peer receives a replica
w	6.1	The vector formed from w_u for all $u \in U$
$x = \|R(q)\|$	2.3	The number of replicas of a query (blue ball)
X		A generic random variable used with many definitions
$y = \|R(d)\|$	2.3	The number of replicas of a piece of data (red ball)
Y		A generic random variable used with many definitions
Z		A generic random variable used with many definitions

Bibliography

[1] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-Grid: a Self-Organizing Structured P2P System. *SIGMOD Record*, 32:29–33, September 2003.

[2] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. Multifaceted simultaneous load balancing in dht-based p2p systems: A new game with old balls and bins. In *Self-star properties in complex information systems*, pages 373–391. Springer, 2005.

[3] Divyakant Agrawal and Amr El Abbadi. Efficient solution to the distributed mutual exclusion problem. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 193–200. ACM, 1989.

[4] Yair Amir, Louise E Moser, Peter M Melliar-Smith, Deborah A Agarwal, and Paul Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems (TOCS)*, 13(4):311–342, 1995.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[6] Asad Awan, Ronaldo A Ferreira, Suresh Jagannathan, and Ananth Grama. Distributed uniform sampling in unstructured peer-to-peer networks. In *System Sciences, 2006. HICSS'06. Proceedings of the 39th Annual Hawaii International Conference on*, volume 9, pages 223c–223c. IEEE, 2006.

[7] W-T Balke, Wolfgang Nejdl, Wolf Siberski, and Uwe Thaden. Progressive distributed top-k retrieval in peer-to-peer networks. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 174–185. IEEE, 2005.

[8] Mauricio Barahona and Louis M Pecora. Synchronization in small-world systems. *Physical review letters*, 89(5):054101, 2002.

[9] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[10] Krista Bennett, Christian Grothoff, Tzvetan Horozov, Ioana Patrascu, and Tiberiu Stef. Gnunet-a truly anonymous networking infrastructure. In *Proc. Privacy Enhancing Technologies Workshop (PET)*, 2002.

[11] Ranjita Bhagwan, George Varghese, and Geoffrey M Voelker. *Cone: Augmenting DHTs to support distributed resource discovery*. Department of Computer Science and Engineering, University of California, San Diego, 2003.

[12] Silvia Bianchi, Sabina Serbu, Pascal Felber, and Peter Kropf. Adaptive load balancing for dht lookups. In *Computer Communications and Networks, 2006. ICCCN 2006. Proceedings. 15th International Conference on*, pages 411–418. IEEE, 2006.

[13] Kenneth P Birman, Robbert Van Renesse, et al. *Reliable distributed computing with the Isis toolkit*, volume 85. IEEE Computer Society Press Los Alamitos, 1994.

[14] Béla Bollobás. *Random Graphs*. Cambridge University Press, 2nd edition, 2001.

[15] Angela Bonifati, Ugo Matrangolo, Alfredo Cuzzocrea, and Mayank Jain. XPath Lookup Queries in P2P Networks. In *Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management (WIDM'04)*, pages 48–55, New York, NY, USA, 2004. ACM Press.

[16] Stephen Poythress Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[17] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-Like P2P Systems Scalable. In *Proceedings of ACM SIGCOMM'03*, pages 407–418, New York, NY, USA, 2003. ACM Press.

[18] Stuart Cheshire, Marc Krochmal, and Kiren Sekar. Nat port mapping protocol (nat-pmp). *draft-cheshire-nat-pmp-03 (work in progress)*, 2008.

[19] Shun Yan Cheung, Mostafa H. Ammar, and Mustaque Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):582–592, 1992.

[20] Tae Woong Choi and P. Oscar Boykin. Deetoo: Scalable Unstructured Search Built on a Structured Overlay. In *Proceedings of the International Workshop on Hot Topics in Peer-to-Peer Systems (HOTP2P'10)*, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[21] Tom Chothia and Konstantinos Chatzikokolakis. A survey of anonymous peer-to-peer file-sharing. In *Embedded and Ubiquitous Computing–EUC 2005 Workshops*, pages 744–755. Springer, 2005.

[22] Fan Chung and Linyuan Lu. The Volume of the Giant Component of a Random Graph with Given Expected Degrees. *SIAM Journal on Discrete Mathematics*, 20(2):395–411, 2007.

[23] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2001.

[24] Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 177–190. ACM, 2002.

[25] Vasilios Darlagiannis, Andreas Mauthe, and Ralf Steinmetz. Sampling cluster endurance for peer-to-peer based content distribution networks. *Multimedia systems*, 13(1):19–33, 2007.

[26] David Dittrich. So you want to take over a botnet. In *Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats*, pages 6–6. USENIX Association, 2012.

[27] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR)*, 35:114–131, June 2003.

[28] Ronaldo A. Ferreira, Murali Krishna Ramanathan, Asad Awan, Ananth Grama, and Suresh Jagannathan. Search with Probabilistic Guarantees in Unstructured Peer-to-Peer Networks. In *Proceedings of P2P'05*, pages 165–172, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Joel Friedman. A proof of alon's second eigenvalue conjecture. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 720–724. ACM, 2003.

[30] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM (JACM)*, 32(4):841–860, 1985.

[31] Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 134–139. ACM, 2004.

[32] David K Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM, 1979.

[33] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[34] Sarunas Girdzijauskas, Wojciech Galuba, Vasilios Darlagiannis, Anwitaman Datta, and Karl Aberer. Fuzzynet: Ringless routing in a ring-like structured overlay. *Peer-to-Peer Networking and Applications*, 4(3):259–273, 2011.

[35] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random Walks in Peer-to-Peer Networks. In *Proceedings of the Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, volume 1, March 2004.

[36] Geoffrey Grimmett and David Stirzaker. *Probability and random processes*. Oxford university press, 2001.

[37] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer

File-Sharing Workload. *SIGOPS Operating System Review*, 37:314–329, October 2003.

[38] Jani Hautakorpi and Göran Schultz. A Feasibility Study of an Arbitrary Search in Structured Peer-to-Peer Networks. In *Proceedings of 19th International Conference on Computer Communications and Networks (ICCCN'10)*, pages 1–8. IEEE, August 2010.

[39] Sandra M Hedetniemi, Stephen T Hedetniemi, and Arthur L Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18(4):319–349, 1988.

[40] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems (TOCS)*, 4(1):32–53, 1986.

[41] M Frans Kaashoek and David R Karger. Koorde: A simple degree-optimal distributed hash table. In *Peer-to-Peer Systems II*, pages 98–107. Springer, 2003.

[42] Gene Kan. Gnutella. In Andy Oram, editor, *Peer-to-Peer - Harnessing the Power of Disruptive Technologies*, pages 62–79. OŔeilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2001.

[43] Sebastian Kaune, Konstantin Pussep, Christof Leng, Aleksandra Kovacevic, Gareth Tyson, and Ralf Steinmetz. Modelling the Internet Delay Space Based on Geographical Locations. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (ICPADS'09)*, pages 301 – 310, February 2009.

[44] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-Based Computation of Aggregate Information. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03)*, pages 482–491, Washington, DC, USA, 2003. IEEE Computer Society.

[45] Jon Kleinberg. The small-world phenomenon: an algorithm perspective. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 163–170. ACM, 2000.

[46] Predrag Knežević, Andreas Wombacher, and Thomas Risse. Dht-based self-adapting replication protocol for achieving high data availability. In *Advanced Internet Based Systems and Applications*, pages 201–210. Springer, 2009.

[47] Max Lehn, , Tonio Triebel, Christof Leng, Alejandro Buchmann, and Wolfgang Effelsberg. Performance evaluation of peer-to-peer gaming overlays. In *Proceedings of the IEEE Tenth International Conference on Peer-to-Peer Computing (P2P'10)*, pages 1–2. IEEE, 2010. demo.

[48] Max Lehn, Christof Leng, Robert Rehner, Tonio Triebel, and Alejandro Buchmann. An online gaming testbed for peer-to-peer architectures. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 474–475. ACM, 2011. demo.

[49] Christof Leng. *BubbleStorm: Replication, Updates, and Consistency in Rendezvous Information Systems*. PhD thesis, Technische Universität Darmstadt, 2011.

[50] Christof Leng and Wesley W. Terpstra. Distributed SQL Queries with BubbleStorm. In Kai Sachs, Ilia Petrov, and Pablo Guerrero, editors, *From Active Data Management to Event-Based Systems and More*, volume 6462 of *Lecture Notes in Computer Science*, pages 230–241. Springer, nov 2010.

[51] Christof Leng, Wesley W. Terpstra, Bettina Kemme, Wilhelm Stannat, and Alejandro P. Buchmann. Maintaining Replicas in Unstructured P2P Systems. In *Proceedings of the ACM CoNEXT Conference (CoNEXT'08)*, pages 19:1–19:12, New York, NY, USA, 2008. ACM.

[52] Jinyang Li, Boon Loo, Joseph Hellerstein, M. Frans Kaashoek, David Karger, and Robert Morris. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.

[53] Mamoru Maekawa. An algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems (TOCS)*, 3(2):145–159, 1985.

[54] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[55] Dahlia Malkhi, Michael K Reiter, Avishai Wool, and Rebecca N Wright. Probabilistic quorum systems. *Information and Computation*, 170(2):184–206, 2001.

[56] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[57] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems (IPTPS'01)*, pages 53–65, London, UK, 2002. Springer-Verlag.

[58] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. Minerva: A scalable efficient peer-to-peer search engine. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 60–81. Springer-Verlag New York, Inc., 2005.

[59] ApS MOSEK. The mosek optimization tools version 3.2 user's manual and reference, 2002.

[60] Damon Mosk-Aoyama and Devavrat Shah. Computing separable functions via gossip. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 113–122. ACM, 2006.

[61] Athicha Muthitacharoen, Robert Morris, Thomer M Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44, 2002.

[62] Joel H. Spencer N. Alon. *The Probabilistic Method*, chapter Eigenvalues and Expanders. John Wiley & Sons, 3rd edition, 2011.

[63] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

[64] Chris Newman. *SQLite (Developer's Library)*. Sams, Indianapolis, IN, USA, 2004.

[65] Costin Raiciu. *ROAR: Increasing the Flexibility and Performance of Distributed Search*. PhD thesis, University College London, London, UK, 2011.

[66] Costin Raiciu, Felipe Huici, Mark Handley, and David S. Rosenblum. ROAR: Increasing the Flexibility and Performance of Distributed Search. In *Proceedings of SIGCOMM'09*, pages 291–302, New York, NY, USA, 2009. ACM.

[67] Costin Raiciu, David S. Rosenblum, and Mark Handley. Distributed Online Filtering. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'07)*, pages 15–16, New York, NY, USA, August 2007. ACM.

[68] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. *SIGCOMM Computer Communication Review*, 31:161–172, August 2001.

[69] Patrick Reynolds and Amin Vahdat. Efficient Peer-to-Peer Keyword Searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware (Middleware'03)*, pages 21–40, New York, NY, USA, 2003. Springer-Verlag New York.

[70] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[71] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking*, MMCN, page 152, 2002.

[72] Nima Sarshar, P. Oscar Boykin, and Vwani P. Roychowdhury. Percolation Search in Power Law Networks: Making Unstructured Peer-to-Peer Networks Scalable. In *Proceedings of IEEE P2P'04*, pages 2–9, Washington, DC, USA, 2004. IEEE Computer Society.

[73] Pyda Srisuresh and Kjeld Egevang. Traditional ip network address translator (traditional nat), 2001.

[74] Ralf Steinmetz and Klaus Wehrle. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer, Heidelberg, Germany, 2005.

[75] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 149–160. ACM, 2001.

[76] Daniel Stutzbach, Reza Rejaie, and Subhabrata Sen. Characterizing Unstructured Overlay Topologies in Modern P2P File-Sharing Systems. *IEEE/ACM Transactions on Networking*, 16:267–280, April 2008.

[77] Wesley W. Terpstra. Distributed Cartesian Product. Diploma Thesis, Technische Universität Darmstadt, May 2006.

[78] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Jussi Kangasharju, and Alejandro Buchmann. Bit Zipper Rendezvous—Optimal Data Placement for General P2P Queries. In *Proceedings of the EDBT 04 Workshop on Peer-to-Peer Computing & DataBases*, March 2004.

[79] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. BubbleStorm: Resilient, Probabilistic, and Exhaustive Peer-to-Peer Search. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'07)*, pages 49–60, New York, NY, USA, August 2007. ACM.

[80] Wesley W. Terpstra, Christof Leng, and Alejandro P. Buchmann. Brief Announcement: Practical Summation via Gossip. In *Proceedings of the Twenty-Sixth Annual ACM Aymposium on Principles of Distributed Computing (PODC'07)*, pages 390–391, New York, NY, USA, August 2007. ACM.

[81] Wesley W. Terpstra, Christof Leng, and Alejandro P. Buchmann. BubbleStorm: Analysis of Probabilistic Exhaustive Search in a Heterogeneous Peer-to-Peer System. Technical Report TUD-CS-2007-2, Technische Universität Darmstadt, Fachbereich Informatik, Darmstadt, Germany, May 2007.

[82] Wesley W. Terpstra, Christof Leng, Max Lehn, and Alejandro P. Buchmann. Channel-based Unidirectional Stream Protocol (CUSP). In *Proceedings of the IEEE INFOCOM Mini Conference*, March 2010.

[83] Robert H Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, 1979.

[84] Robbert Van Renesse, Kenneth P Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS)*, 21(2):164–206, 2003.

[85] Robbert Van Renesse and Adrian Bozdog. Willow: Dht, aggregation, and publish/subscribe in one protocol. In *Peer-to-Peer Systems III*, pages 173–183. Springer, 2005.

[86] Robert J Vanderbei. Loqo: An interior point code for quadratic programming. *Optimization methods and software*, 11(1-4):451–484, 1999.

[87] Yong Yang, Rocky Dunlap, Michael Rexroad, and Brian F. Cooper. Performance of Full Text Search in Structured and Unstructured Peer-to-Peer Systems. In *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM'06)*, pages 1–12, April 2006.

[88] Yingwu Zhu and Yiming Hu. Efficient, proximity-aware load balancing for dht-based p2p systems. *Parallel and Distributed Systems, IEEE Transactions on*, 16(4):349–361, 2005.